SainSmart Uno
Starter Kits Manual

Version 1.1

Table of Contents

Chapter 1: Arduino Introduction

Overview

Welcome to the wonderful world of Arduino! With the advent of the Arduino microcontroller based boards (Uno, Mega, etc.), their easy access and programmable input/output (I/O) ports, and user friendly program development environment, has helped spark a new movement of inspiration for hobbyists, tinkerers, inventors, and artists alike. Individuals can easily create systems that are capable of sensing its environment, make decisions based on data gathered, and take action accordingly. With Arduino and Arduino compatibles, the sky is the limit as to what you can build.

What exactly is Arduino?

As defined by Wikipedia:

> ***Arduino*** *is a single-board microcontroller to make using electronics in multidisciplinary projects more accessible. The* ***hardware*** *consists of an open-source hardware board designed around an 8-bit Atmel AVR microcontroller, or a 32-bit Atmel ARM. The* ***software*** *consists of a standard programming language compiler and a boot loader that executes on the microcontroller.*

Hardware

The open-source hardware license allows for companies and individuals to make their own replicas or variations of this base design from schematics, parts lists, etc., which are available for free from the Arduino website, see www.arduino.cc. At the heart of the Arduino Uno board is an Atmel 8-bit microcontroller, the Atmega328. The main components of the Arduino and SainSmart Uno R3 will be covered shortly.

Software

Arduino programs, which are known as sketches, are written in the C/C++ programming language using the integrated development environment (IDE). Thus, if you know C, you have a great head start. If you don't know C or any other programming languages, not to worry. This manual is designed to slowly introduce you to the Arduino programming language by giving detailed example code for each new concept covered, along with a brief explanation. So, if you already know C, you

can either skim through or skip the code explanation sections. However, the examples use some unique functions and libraries that were originally written for Wiring or developed specifically for Arduino, so these will most likely be new to all first time Arduino programmers, as well as many experienced Arduino and C programmers.

Integrated development environment (IDE)

The Arduino's integrated development environment (IDE) is Java based, thus making it a cross-platform application (capable of running on different operating systems without need for re-compilation or modification of the code). The IDE is free to download from the Arduino website and is also open-source. As you will see in the first chapter, it greatly simplifies the process of interfacing to, and programming the Arduino. When you launch the IDE, it will appear as shown in Figure 1.1.

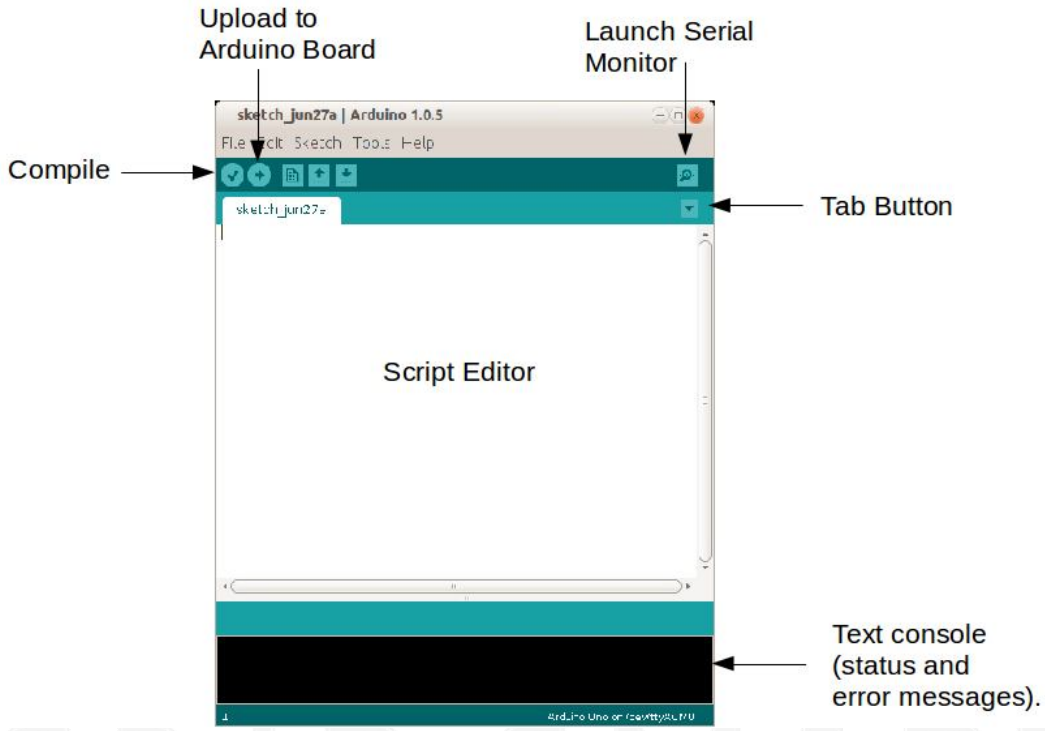


Figure 1.1 The Arduino IDE.

What is a microcontroller?

A microcontroller is essentially a small computer on a single integrated circuit (IC) or "chip". Besides having a central processing unit (CPU), which runs your program, it has input/output (I/O) circuitry built into the chip as well.

Arduino Uno R3

The Arduino Uno is a microcontroller board based on the Atmel ATmega328

microcontroller. The board has 14 digital input/output pins (of which 6 can be used as PWM outputs), 6 analog inputs, a 16 MHz ceramic resonator, a USB connection, a power jack, an ICSP header, and a reset button. It contains everything needed to support the microcontroller; simply connect it to a computer with a USB cable. The USB cable will supply the Uno with a regulated 5v DC input voltage. Once your Uno has been programmed and if it doesn't need the USB connector for communications, then external power from an AC-to-DC adapter or a 9v battery (7v to 12v DC input at the barrel connector) can be used, see Figure 1.2.

Note: Never connect external power to the Uno if it is connected to a PC with the USB connector. This may damage the Uno and/or your PC.

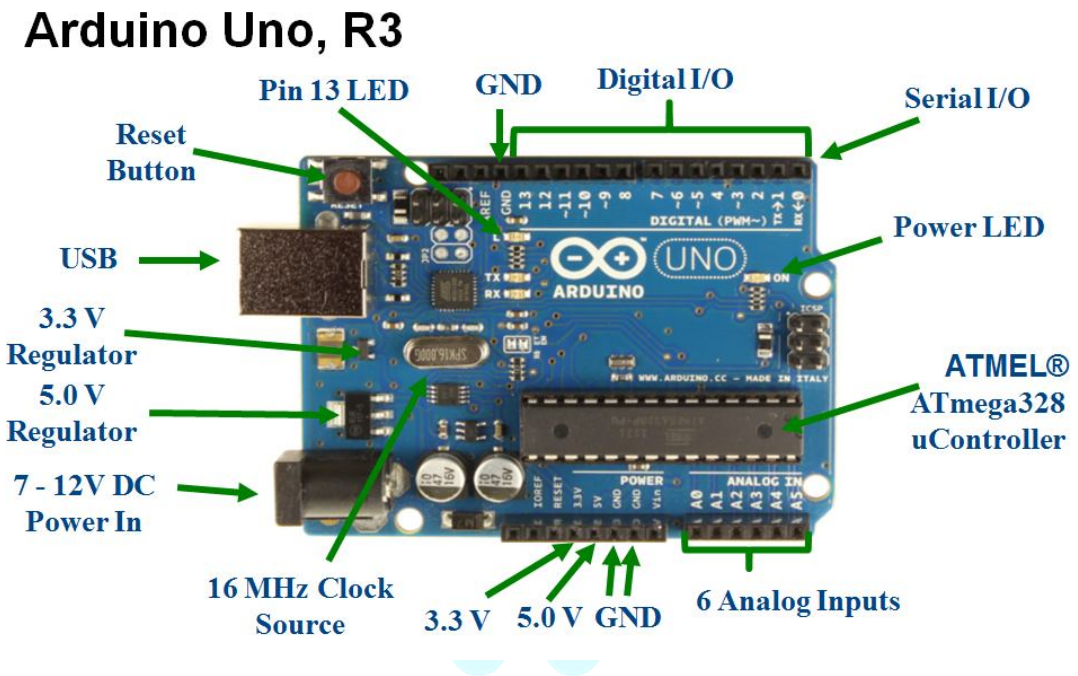


Figure 1.2 Arduino Uno, R3 board.

SainSmart Uno R3

The SainSmart Uno R3 is an Arduino-compatible Uno R3. It is replicated from the files found in the Arduino open source hardware library. It has the same pinouts, parts, and functionality as the Arduino Uno R3. In addition, the SainSmart version has the added ability to work at 5v or 3.3v at the flip of an onboard switch. This is very useful due to the fact that Arduino based products are migrating to 3.3v, due to the increasing use of ARM processors, which only operate at 3.3v. Thus, whenever buying shields (boards that plug on top of the Uno for added functionality), make sure that the Uno and shield are working at the same I/O voltage levels. Otherwise, the microcontroller and/or shield components may get damaged.

Additionally, the SainSmart Uno R3 has extra male pins for the digital and analog I/O ports (S). Next to each of these pins is a working voltage level pin (V) and a ground pin (G). This is very helpful for certain connector types that you may want to use with your board, see Figure 1.3.

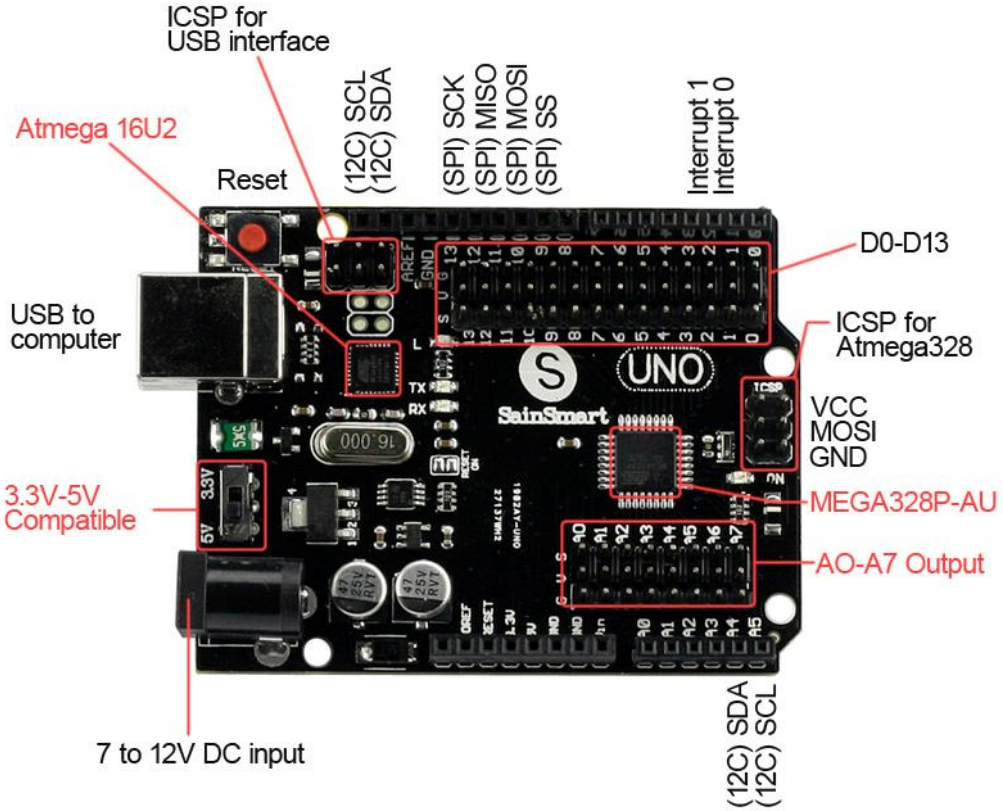


Figure 1.3 SainSmart Uno, R3.

Suggested Reference and Reading Material

There are many excellent books available on programming the Arduino, which typically covers the basics of the C programming language, as well as basic electronics and circuit theory. Additionally, the Arduino Website offers many examples, tutorials, and links to other related material on basic electronics and Arduino programming. The following lists some of the recommended books on this topic, as well as some comments on their contents:

Intro-level:

The following is a good starter book for learning the basics of writing sketches. This is accomplished by focusing mainly on the C programming language along with a few simple circuits.

- **Programming Arduino: Getting Started with Sketches** by Simon Monk. Published by McGraw Hill (2012).

The following book is a great introductory book for Arduino programming, using the IDE, computer-Arduino communication, electronics basics, etc. It is much more comprehensive than the previously listed book.

- **Beginning Arduino Programming** by Brian Evans. Published by Apress; 1 edition (October 17, 2011).

More advanced:

These two books are very comprehensive and come with access to all sketch examples (which are bountiful!) in the books. They both assume a certain level of knowledge in programming and electronics.

- **Arduino Cookbook** (2nd Edition) by Michael Margolis. Published by O'Reilly (Dec 30, 2011).

- **Exploring Arduino: Tools and Techniques for Engineering Wizardry** by Jeremy Blum. Published by John Wiley & Sons, Inc. (2013).

Other books recommended under the arduino.cc website

Additional recommended reference material, some of which are free, can be found at: http://playground.arduino.cc/Main/ManualsAndCurriculum

Basic Contents of this Manual

Although this manual's main focus is on presenting example applications of the electronic components that come in the SainSmart line of Arduino Uno Compatible Starter Kits, it also covers the basic knowledge needed in programming and electronics such that the beginner understands why and how these applications work. This is accomplished by supplying functionality details of these electronic components, and covering example applications to illustrate hardware connectivity and corresponding software code, along with explanation of newly presented

concepts.

System Setup Requirements

This manual assumes that the Arduino IDE software has been downloaded from the official Arduino website: http://arduino.cc/en/Guide/HomePage, and communication between the SainSmart Uno board and your computer has been verified. Also, tips on troubleshooting some common errors can be found on the above Website, under the Troubleshooting link. The first few sketches and chapters of this manual will also be directed at testing the communication link to, and basic functionality of the SainSmart board. They will also cover some of the basics of programming/sketch writing.

Chapter 2: First Sketch – Blink

The Uno comes preloaded with the blink sketch. So, when you first connect your Uno to your PC via the accompanying USB cable, the light emitting diode (LED) next to pin 13 will start to blink. The code for this sketch can be found under the following section of the IDE: File->Examples->Basics->Blink. Just click on Blink and the program/sketch will open in a new window, as shown in Figures 2.1 and 2.2.

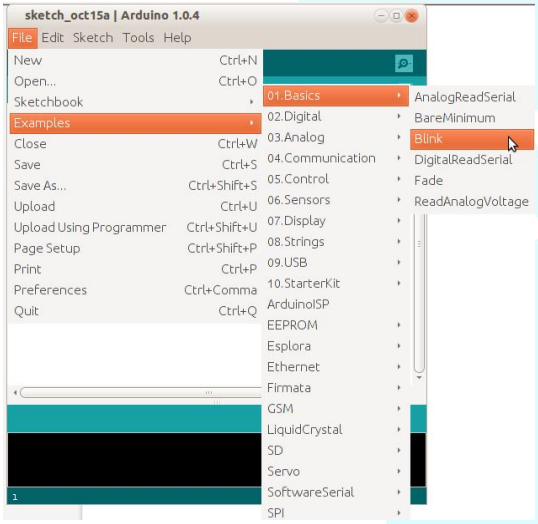


Figure 2.1 Opening the Blink sketch.

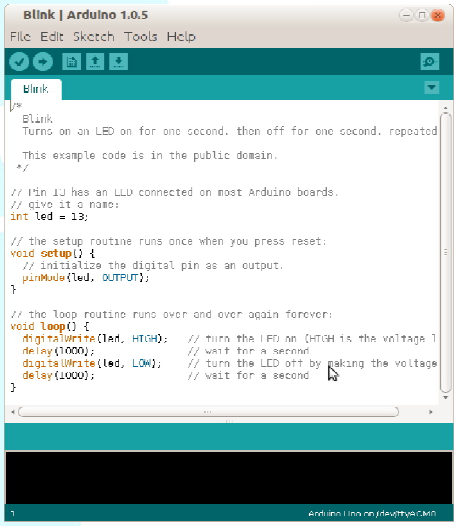


Figure 2.2 The Blink sketch in a new window.

Before we cover how this sketch works, we will first review the basic parts of any sketch, as shown in Figure 2.3. For a larger view of the blink sketch, see Figure 2.4.

Anatomy of an Arduino Sketch

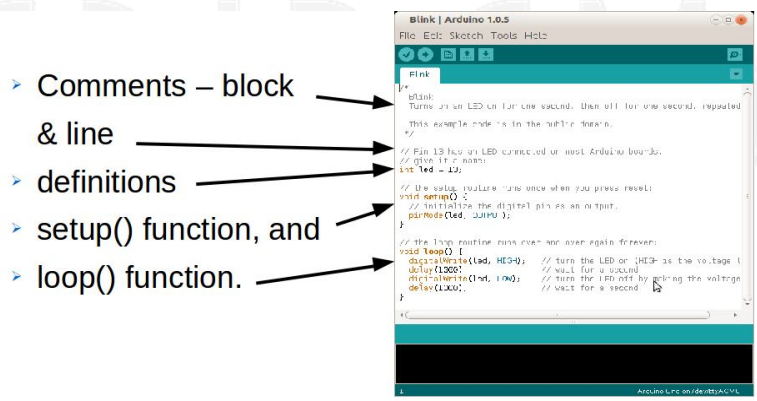


Figure 2.3 The four basic parts of an Arduino sketch.

Comments

The comment sections are optional and are ignored by the compiler. However, it is good coding technique to have plenty of comments. This makes your code easier to read and debug. Also, it is a great memory jogger when you have been away from your sketch for a while!

Comments can be either inline or block format. With inline comments, everything to the right of the // and to the end of the line is ignored by the compiler. With block comments, everything between the start of the comment: /* and the end of the comment: */ is ignored. A block comment can span multiple lines, so be careful to add the end of comment indicator, or the rest of your code with be ignored and you will most likely get a compile error. Also make sure to update your comments if you go back and change your code.

Definitions and Data Types

The following are types of declarations that you would normally place in the definitions section, which is considered anywhere before the setup() function:

- Libraries/header files to include in the sketch.
  Example: #include <filename.h>,
- User/programmer defined function declarations.
  Example: void myFunction ();
- Global variables (e.g., int, float, arrays, structures, etc.), and
- Global constants, which are values that won't and can't change, such as pin definitions, etc.

The various data types found in the definitions section are listed in Table 1. In addition to this list, there are two data type modifiers: ***const*** and ***unsigned***. The definitions and example usages are as follows:

const Declares a variable as constant, which means that its value cannot be modified in the sketch. Example: **const float** PI = 3.14;

unsigned A modifier for integer values, designating that they represent only non-negative values. This also increases the maximum value the variable can store, see Table 1. Example: **unsigned int** myPosNumber = 3;

Type	Memory (bytes)	Range	Notes
boolean	1	true or false (1 or 0)	
char	1	-128 to +127	Used to represent an ASCII character code; e.g., *A* is represented as 65. Its negative numbers are not normally used.
byte	1	0 to 255	Often used for communicating serial data, as a single unit of data.
int	2	-32768 to +32767	
unsigned int	2	0 to 65535	Can be used for extra precision where negative numbers are not needed. Use caution, as arithmetic with **int**s may cause unexpected results.
long	4	-2,147,483,648 to +2,147,483,647	Needed only for representing very large numbers.
unsigned long	4	0 to 4,294,967,295	See **unsigned int**.
float	4	-3.4028235E+38 to +3.4028235E+38	
double	4	Same as float	Normally, this would be 8 bytes and higher precision than **float**, with a greater range. However, on Arduino, it is the same as **float**.

Table 2.1 Data Types.

Setup Function

The ***setup()*** function is run first and only once at power up or after reset. This is where commands that perform some type of initialization is located (e.g., pin modes, communication initialization, etc.). This function accepts no arguments and returns no value.

Loop Function

The code in the ***loop()*** function is run over and over again, indefinitely (while the board has power anyway). So, this is where the "brains" of your sketch will be located.

Now let's look at the details of the blink sketch, so that we understand what is going on and why.

Blink Sketch

```
/*
  Blink
  Turns on an LED on for one second, then off for one second, repeated

  This example code is in the public domain.
 */

// Pin 13 has an LED connected on most Arduino boards.
// give it a name:
int led = 13;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage l
  delay(1000);               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off by making the voltage
  delay(1000);               // wait for a second
}
```

Figure 2.4 The blink sketch.

The comment section at the top of the sketch, gives the name of the sketch and a brief description of its functionallity. This is good programming practice and should be included in all of your sketches.

Next, a variable named ***led*** is declared of type ***int*** (integer – see Table 1 for the range of values for this type of variable). The variable, led, is initialized to 13, which is the the Uno's data I/O pin number we wish to use. Pin 13 of the Uno is the only I/O port that has an LED connected directly to it. There is also a resister in series with this LED, which is used to limit the current that the pin will need to supply. This is important, because the I/O pin could possibly burn out if it is sourcing or sinking too much current. This topic will be covered in a later chapter.

The setup() funtion calls another function named ***pinMode()*** to initialize pin 13 to type "OUTPUT". The pinMode() function is located in the Arduino standard library (Arduino.h), which is automatically included elsewhere (main.cpp, which is hidden from the programmer in the IDE), so we didn't have to include it in the declarations section. As you can see, the pinMode() function accepts two arguments: the first is the I/O port or pin number, and the second is the type of port it is being set to, "INPUT" or "OUTPUT". After setup() is run, pin 13 of the microcontroller is initialized to be an output port.

Now for the "brains" of the sketch. In the loop() function, the pin 13 output port is

alternating between being set to a logic 0 or “LOW” (0v), and a logic 1 or “HIGH” (+5v or +3.3v, depending at which voltage your board is set to run at) by use of the function ***digitalWrite()***. This function accepts two arguments: first the pin number, then the output value of logic “HIGH” or “LOW”. This alternating voltage output value, along with a delay in between, creates the blinking of the LED connected to pin 13. The ***delay()*** function does as it is named, it causes the program to wait a certain length of time before continuing to the next line of code. The argument this function takes is an integer that represents the number of milliseconds (ms) to delay execution of the next line of code. Here, the argument of 1000 creates a delay of 1000 milliseconds, which is equal to 1 second. Therefore, the LED will be on for 1 second and off for 1 second, and it will keep repeating this pattern.

Modifying Blink

You can easily modify this code to make the LED blink faster or slower by either increasing or decreasing the delay values. To make the change noticeable, you should change the delay values by at least a factor of 2, or even better by 4. For example, if the values in the delay() functions are changed from 1000ms to 250ms, then the LED will be on for ¼ of a second and off for ¼ of a second.

Saving Your Sketch

You can either exit out of the blink sketch window without saving your modifications or you can save it by renaming it and placing it into you own sketch folder. This section describes the steps you need to take to save your modified blink sketch.

If you press save under File or via the Save button, you will get a pop-up window, similar to the one shown in Figure 2.5:

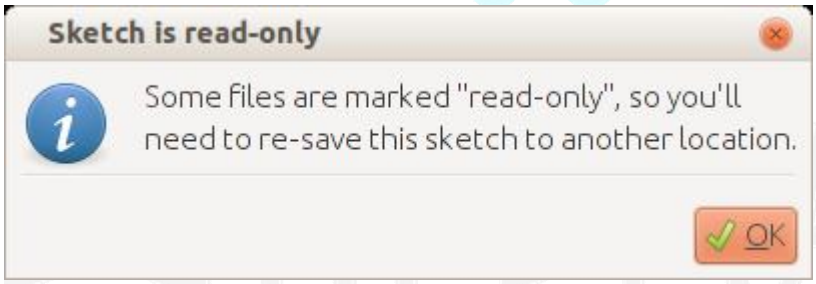


Figure 2.5 Read-only error message.

That is because the files under Examples are read-only. That is so they can be used as a reference without you having to worry if it is the original file or not. So, you are prompted to save the file in a different directory. You can use the default one, or create a special sketch folder of your own, outside the IDE framework. You can also rename the file being saved to something more descriptive, to set it apart from the original, like “myBlink”.

Chapter 3: Hello World!

In this chapter, we will learn to use the Arduino IDE serial interface. We will send data from the SainSmart Uno to your computer, and display the data on the Arduino IDE's Serial Monitor window.

helloWorld Sketch

To get started, you can either type the following code into a blank Arduino sketch, or bring up the BareMinimum sketch under File->Examples->Basics->BareMinimum and type in the code shown in the setup() function, or you can bring up the helloWorld sketch that can be downloaded from the SainSmart Website. If you are using the sketches for the Uno Starter Kits Tutorials and have downloaded them, then from the Arduino IDE go to File->Open-> and work your way down to: /Chapter 3 Hello World!/helloWorld/ and select the helloWorld.ino file. The helloWorld code is shown in Figure 3.1.

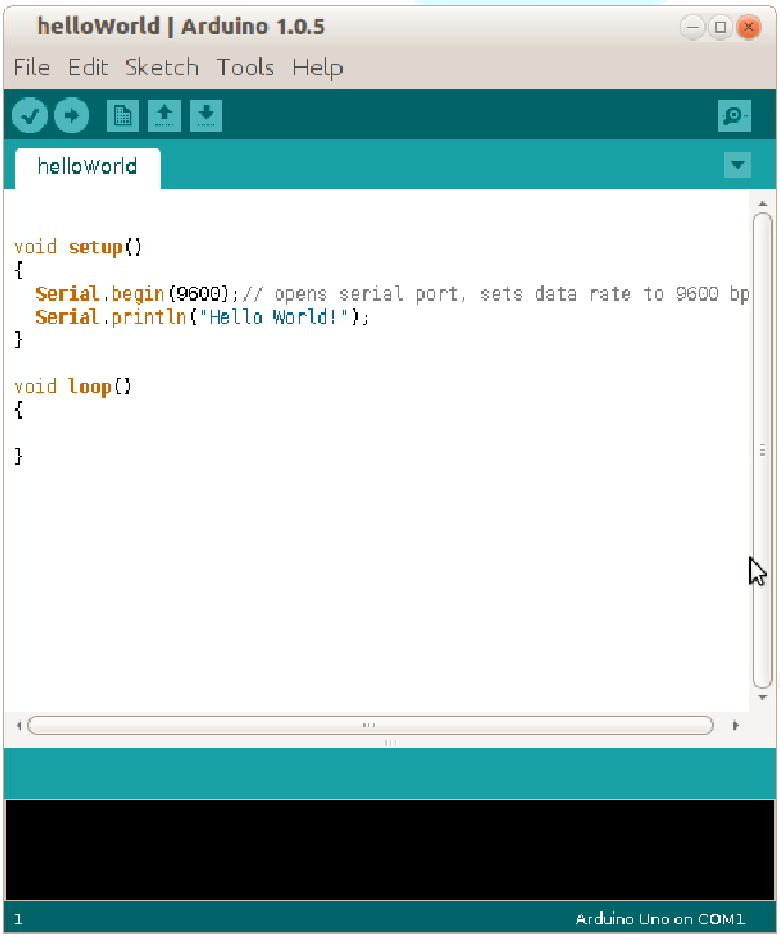


Figure 3.1 helloWorld sketch.

Operation

1. Compile/upload the sketch to your SainSmart Uno board.
2. After a successful upload, click "Serial Monitor" button in the upper right corner of the IDE (see Figure 1.1).
3. Make sure the baud rate in the lower right hand corner of the Serial Monitor window is set to 9600.
4. You should see "Hello World!" printed in the Serial Monitor window.

Sketch Explained

As you can see there is very little to this code. However, there are a few new items that were used. *First*, we used the **Serial** class to set the data rate of our serial connection. *Then* we used the Serial class to send the "Hello World!" string from the Uno board to the computer. *Lastly*, we used the IDE's Serial Monitor window to display the data sent by the Uno to the computer. A brief description of these new concepts follow.

Classes

A class is the basic building block of object oriented programming languages, such as C++. Since the Arduino programming language uses the C/C++ language, classes can be found in many of the Arduino libraries. Essentially, they are a way of grouping related functions and data under one object, the class. The only thing we really need to know about classes is how to use them. In this case, we used the Serial class, which is located in one of Arduino's standard libraries. The Serial.begin() and Serial.println() are special serial communication functions, which are defined in the Serial class.

Serial Communication

The Serial.begin() function takes an integer argument and has to be one of the following values: 300, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200. This is the baud rate, which is equivalent to bits per second (bps) for our application. The value you choose for the argument has to match the value selected in the lower right hand corner of the Serial Monitor window. Otherwise, the computer and the Uno will be "talking" to each other at different rates, causing nothing to appear or strange symbols.

Serial port communication uses digital pins 0 (receive or RX) and 1 (transmit or TX) of the Atmel microcontroller. So, if your sketch uses serial communication to send or receive data from the computer, then you can't use these two digital I/O pins for other functions.

Serial Monitor

As previously mentioned, the Serial Monitor window is launched from the sketch window by pressing the Serial Monitor button in the upper right hand corner, see Figure 1.1 for button location. Additionally, you must have a matching baud rate set in the lower right hand corner to match the Serial.begin() argument. Then, after upload or reset, the string sent by the Uno will appear in this window, see Figure 3.2.

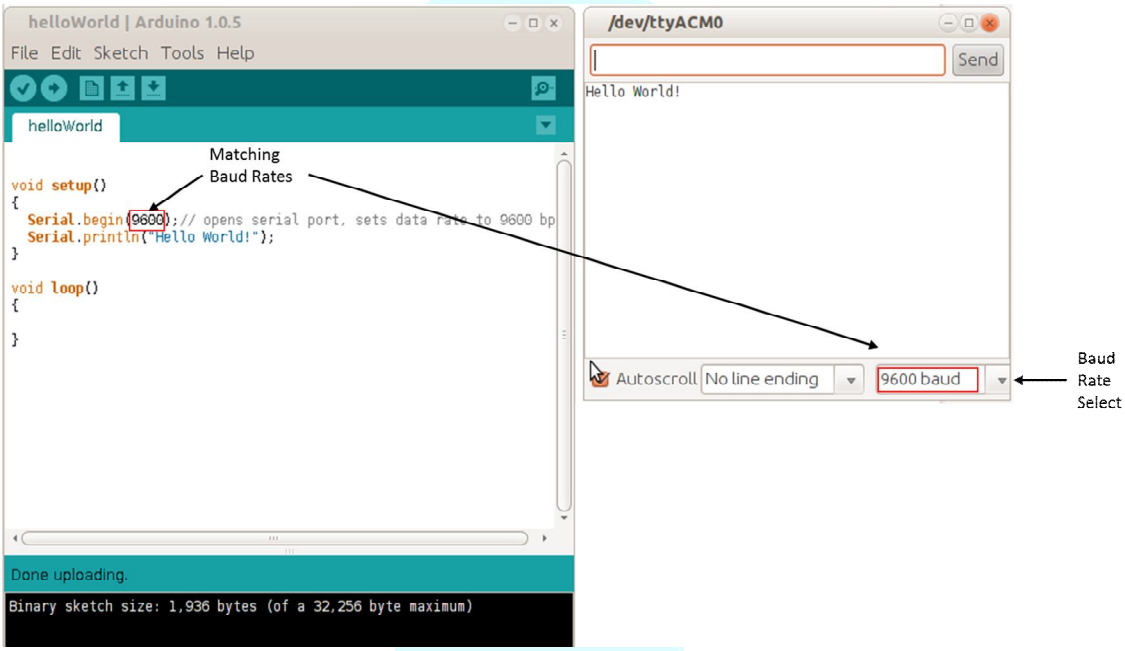


Figure 3.2 The helloWorld sketch window (left) and the Serial Monitor window (right), shown with matching baud rates of 9600.

Chapter 4: LED Blink

Light Emitting Diodes

What is a light emitting diode?

The light emitting diode (LED), as its name indicates, is a diode that emits light! Depending on the type of LED selected, it can either emit in the visible, ultraviolet, or infrared wavelengths. How brightly the LED emits light is dependent on the voltage across its terminals, the amount of current passing through it, as well as the semiconductor material of the LED. The LEDs that we will be using in this sketch emit in the visible spectrum and its color is determined by the type of semiconductor used. For example, a clear LED, as shown in Figure 4.1, can emit a red, green, or yellow color (blue is used in the starter kits in place of the yellow LED) based on its makeup. Some LEDs come with a colored casing, which determines its emitting color. Figure 4.1 shows examples of several types of LEDs available.

Figure 4.1 Visible emitting LEDs in different color casings.

Flashing LED Sketch

Sketch components:

- 1 x LED (any color)
- 1 x 220Ω resistor

- Uno, breadboard, & jumper wires

Connect your circuit as shown in Figure 4.2. The LED will not light if put in with its terminals reversed. That is because the LED, being a diode, will only allow current to flow in one direction. The two LED terminals go by the following names: the anode, which connects to positive voltage supply (longer lead), and the cathode, which connects to ground (shorter lead).

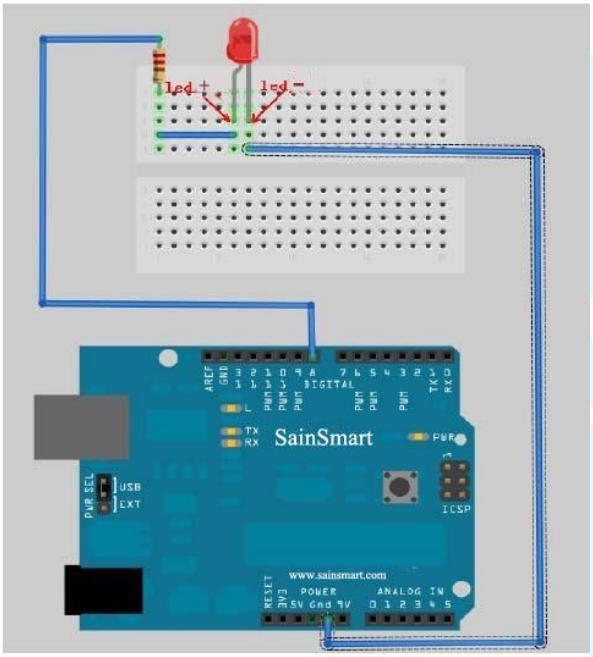


Figure 4.2 Circuit schematic.

Example code from the SainSmart library:

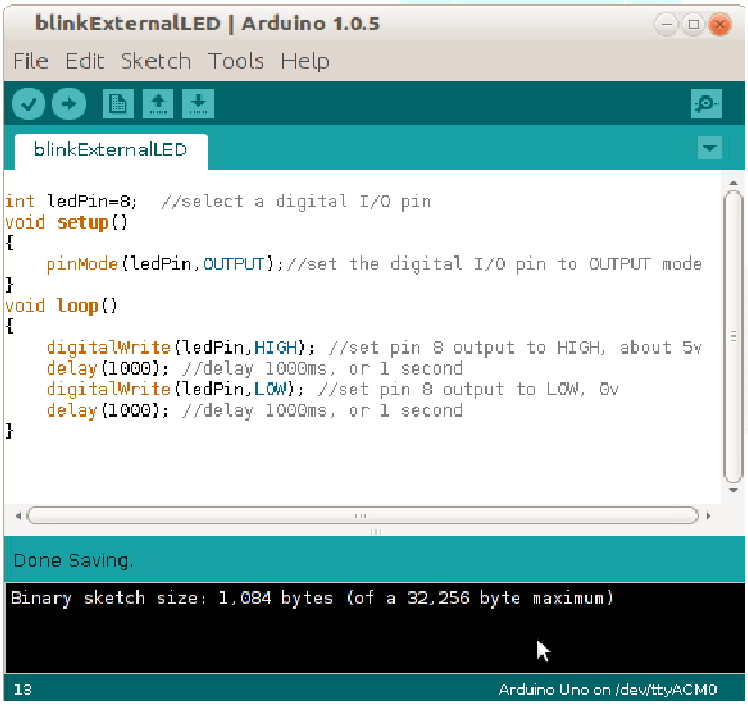


Figure 4.3 External blinking LED sketch.

Sketch Explained

As you can see from Figure 4.3, the code is very similar to the blink program covered in chapter 2. This time, we are using an external LED, which is in series with the 220 ohm resister to limit current. We could have used any of the digital I/O ports/pins for this example. The only change would be in the declaration section, where the integer variable, *ledPin*, is initialized.

You can also use the blink sketch as shown in Figure 2.4, where you would need to connect to pin 13 instead of pin 8. This would give you two blinking LEDs, the one on the PCB board that is connected to pin 13, and the external LED.

Chapter 5: PWM

What is PWM?

Pulse Width Modulation, or PWM, is a technique for getting analog results with digital means. Digital control is used to create a square wave, a signal switched between on and off. This on-off pattern can simulate voltages in between full on (5 Volts) and off (0 Volts) by changing the portion of the time the signal spends on versus the time that the signal spends off. The duration of "on time" is called the pulse width. To get varying analog values, you change, or modulate, that pulse width. If you repeat this on-off pattern fast enough with an LED for example, the result is as if the signal is a steady voltage between 0 and 5v controlling the brightness of the LED.

In Figure 5.1, the green lines represent a regular time period. This duration or period is the inverse of the PWM frequency. In other words, with Uno's PWM frequency at about 500Hz, the green lines would measure 2 milliseconds each. A call to analogWrite() is on a scale of 0 - 255, such that analogWrite(255) requests a 100% duty cycle (always on), and analogWrite(127) is a 50% duty cycle (on half the time) for example.

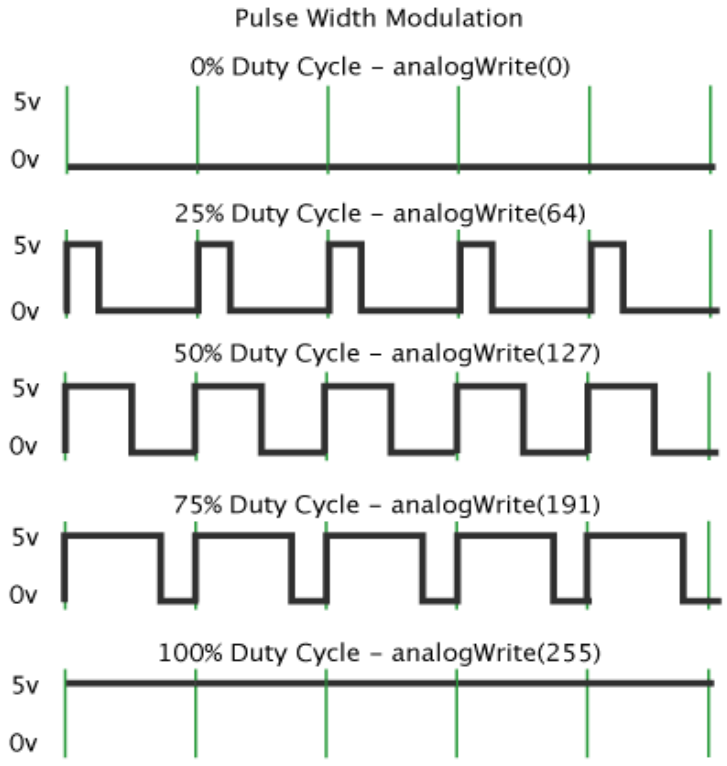


Figure 5.1 Pulse width modulation waveforms.

Arduino analogWrite() Function

As described previously, a call to analogWrite() will generate a steady square wave of the specified duty cycle until the next call to analogWrite() (or a call to digitalRead() or digitalWrite() on the same pin). On most Arduino boards (those with the ATmega168 or ATmega328), this function works on pins 3, 5, 6, 9, 10, and 11. On the Arduino Mega, it works on pins 2 through 13. Older Arduino boards with an ATmega8 only support analogWrite() on pins 9, 10, and 11. The Arduino Due supports analogWrite() on pins 2 through 13, plus pins DAC0 and DAC1. Unlike the PWM pins, DAC0 and DAC1 are Digital to Analog converters, and act as true analog outputs. You do not need to call pinMode() to set the pin as an output before calling analogWrite(). The analogWrite function has nothing to do with the analog pins or the analogRead function.

Syntax
analogWrite(pin, value)

Parameters
pin: The PWM capable pin to write to.
value: The duty cycle: between 0 (always off) and 255 (always on).

Notes and Known Issues
The PWM outputs generated on pins 5 and 6 will have higher-than-expected duty cycles. This is because of interactions with the millis() and delay() functions, which share the same internal timer used to generate those PWM outputs. This will be noticed mostly on low duty-cycle settings (e.g 0 - 10) and may result in the inability to fully turning off the output on pins 5 and 6.

A Fading LED Sketch

Sketch components:

- 1 x 220Ω resistor
- 1 x LED
- Uno, breadboard, & jumper wires

Connect your circuit as illustrated in Figure 5.2.

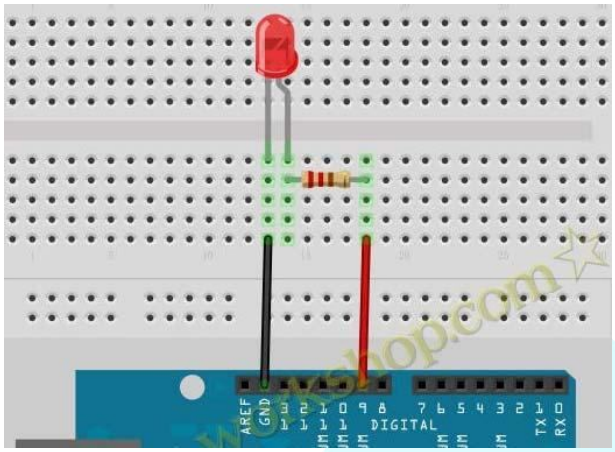


Figure 5.2 Fade circuit schematic.

Example code:

The code shown in Figure 5.3 can be found under File->Examples->Basics->Fade.

Fade | Arduino 1.0.5

File Edit Sketch Tools Help

Fade §

```
/*
 Fade

 This example shows how to fade an LED on pin 9
 using the analogWrite() function.

 This example code is in the public domain.
 */

int led = 9;           // the pin that the LED is attached to
int brightness = 0;    // how bright the LED is
int fadeAmount = 5;    // how many points to fade the LED by

// the setup routine runs once when you press reset:
void setup()  {
  // declare pin 9 to be an output:
  pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop()  {
  // set the brightness of pin 9:
  analogWrite(led, brightness);

  // change the brightness for next time through the loop:
  brightness = brightness + fadeAmount;

  // reverse the direction of the fading at the ends of the fade:
  if (brightness == 0 || brightness == 255) {
    fadeAmount = -fadeAmount ;
  }
  // wait for 30 milliseconds to see the dimming effect
  delay(30);
}
```

Done uploading.

Binary sketch size: 1,316 bytes (of a 32,256 byte maximum)

1 Arduino Uno on /dev/ttyACM0

Figure 5.3 Fade sketch.

Sketch Explained

This sketch declares three variables in the definitions section, as shown at the top of Figure 5.3. First, the PWM pin 9 is selected for driving the LED. Next, a variable named *brightness* is declared and initialized to zero. This variable is used to hold the actual phase value of between 0 and 255, which will be used for the analogWrite() function during each iteration of the loop() function. Lastly, a variable that will hold the PWM phase step increase and decrease amount between iterations of the loop() function, *fadeAmount*, is defined and initialized to 5.

For each iteration of the loop() function, the PWM will increase from 0 to 255 (off to fully on) in steps of *fadeAmount,* or 5 in this case. When the phase value, *brightness*, reaches the maximum value of 255, which is checked by the "if" statement, the *fadeAmount* is set to negative 5. This causes the phase value to count backwards by 5, from 255 to 0, during each iteration of the loop() function. The delay(30), allows for the LED to remain at each PWM value for 30 ms. What happens if you increase or decrease this value?

Chapter 6: Tri-Color LED Module

The tri or three color LED module that comes with the Uno Starter Kits has, as its name implies, three separate, different color LEDs on a single, clear package. The backside of the module is shown in Figure 6.1.

Figure 6.1 Tri-color LED module – backside.

This module works the same as the single LEDs covered in the previous chapters. But, instead of a single anode (connects to positive voltage supply), there are three, one for each color. They all share a common cathode, which connects to ground.

Simple sample sketch!

Sketch components:

- 3 x 220 ohm resistors
- 1 x tri-color LED module
- Uno, breadboard, & jumper wires

This sketch simply rotates through the three colors over and over again. Each LED stays on for two seconds before being turned off and the next color LED in turn is illuminated. The circuit connection is illustrated in Figure 6.2.

Note: Don't use the Uno's pin numbers as shown on the outside of the Uno symbol in the schematic in Figure 6.2. These are the actual pin numbers of the semiconductor package. Instead, use the labels that are inside the box. Example: Digital I/O port 3 is show as (PD3) **DIGITAL3**. This corresponds to **pin 3** on the Uno board/header.

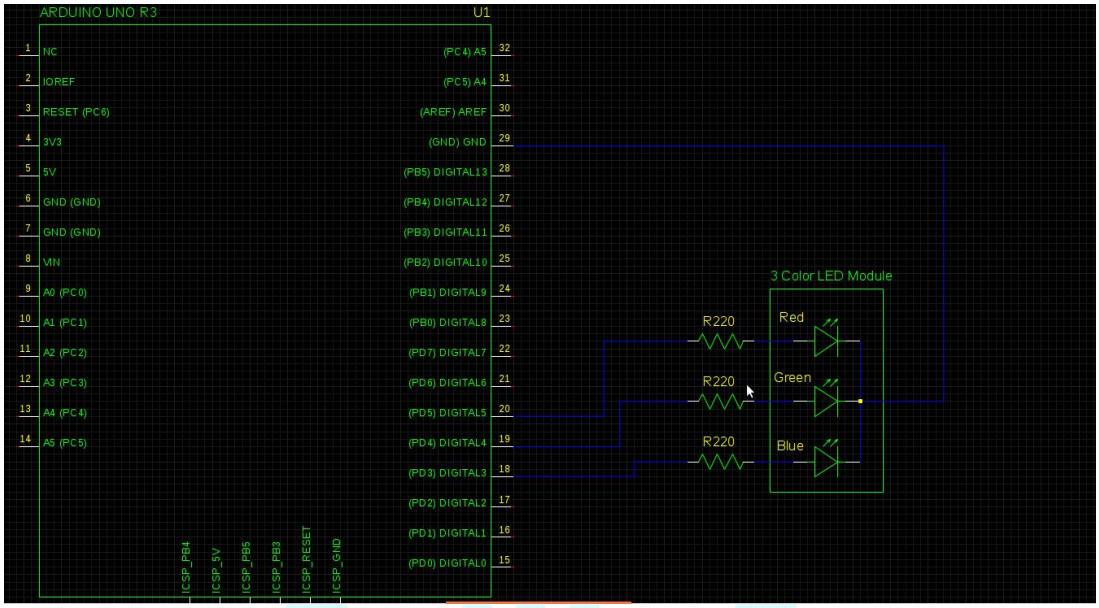


Figure 6.2 Sketch schematic.

Example code from the SainSmart library:

Figure 6.3 shows the code for this sketch. It is very similar to the blink sketch, but instead of turning on and off a single LED, this sketch controls three LED inputs. Additionally, a couple of more advanced programming concepts are used here.

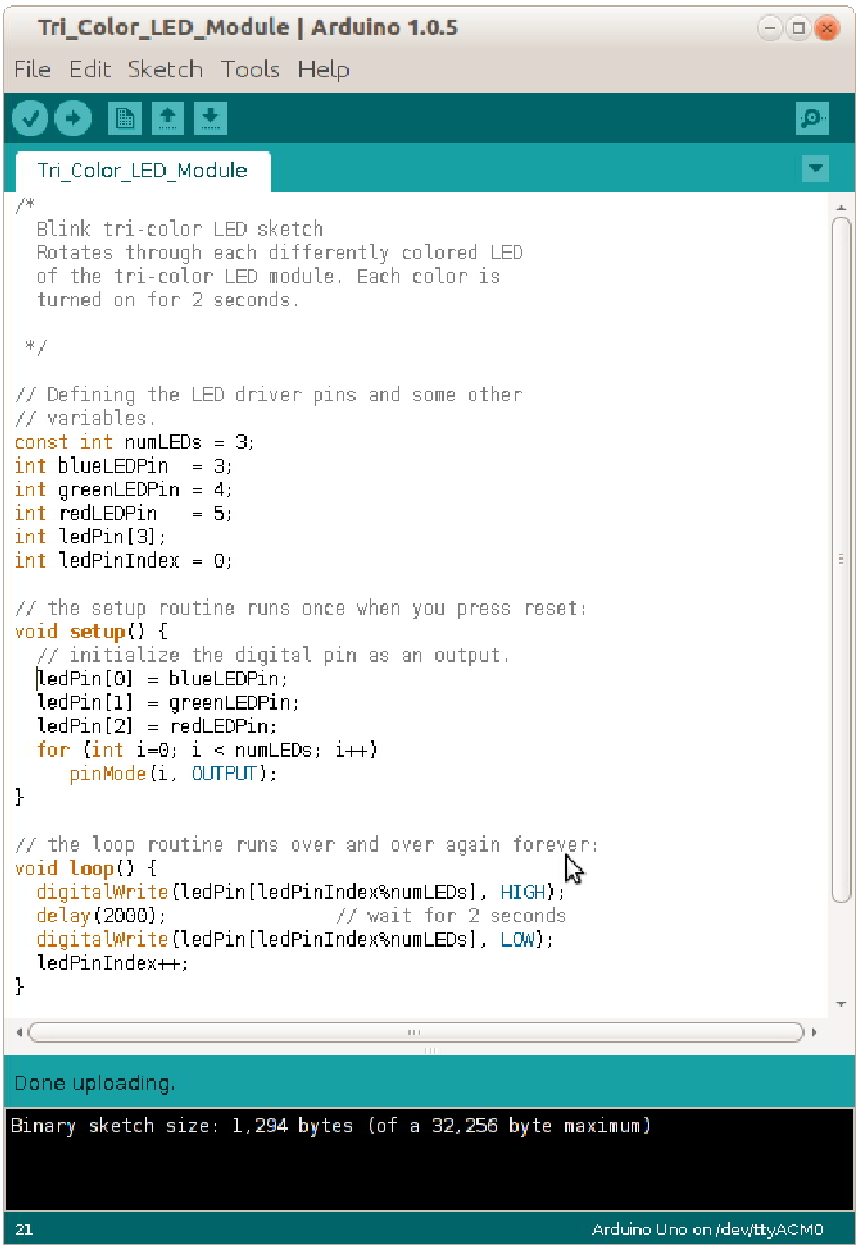


Figure 6.3 Rotating LED display sketch.

Chapter 7: Buzzer

What is a buzzer?

A buzzer is a transducer (converts electrical energy into mechanical energy) that typically operates in the lower portion of the audible frequency range of 20 Hz to 20 kHz. This is accomplished by converting an electric, oscillating signal in the audible range, into mechanical energy, in the form of audible waves. Buzzers are widely used in computers, printers, copiers, alarms, electronic toys, automotive electronic equipment, telephones, cell phones, timers, etc.

Figure 7.1 Buzzer

Additionally, buzzers can be divided into two major categories: *active* and *passive*. An active buzzer only requires an external DC voltage to make it sound. A passive buzzer requires an external AC or oscillating signal in the audio range to drive it. Therefore the tone produced by the active buzzer can't be changed because the circuit that determines its operating frequency is built into the buzzer's package. However, this does save on the need for additional logic to drive the buzzer. Example single tone applications encompass warning or notification indicators, such as a smoke detector alarm or a clothes dryer's done indicator. The tone a passive buzzer makes can be changed by changing the driving signal's frequency. This flexibility does come at a price though, in the form of additional circuitry and processor time.

The Uno Starter Kits come with 1 of each type of buzzer. The slightly taller one, with a solid, plastic bottom coating, is the active buzzer. The buzzer with an open bottom, where you can see the terminals connected to a printed circuit board (PCB), is the passive buzzer. The next few sections illustrate some example applications for each of these buzzer types.

Passive buzzer sketches

Sketch components:

- 1x Passive Buzzer
- Uno, breadboard, & jumper wires

The passive buzzer is connected as shown in Figure 7.2 for the following sketch. The positive terminal of the buzzer is connected to the Uno's digital I/O pin 7 and the negative terminal is connected to ground.

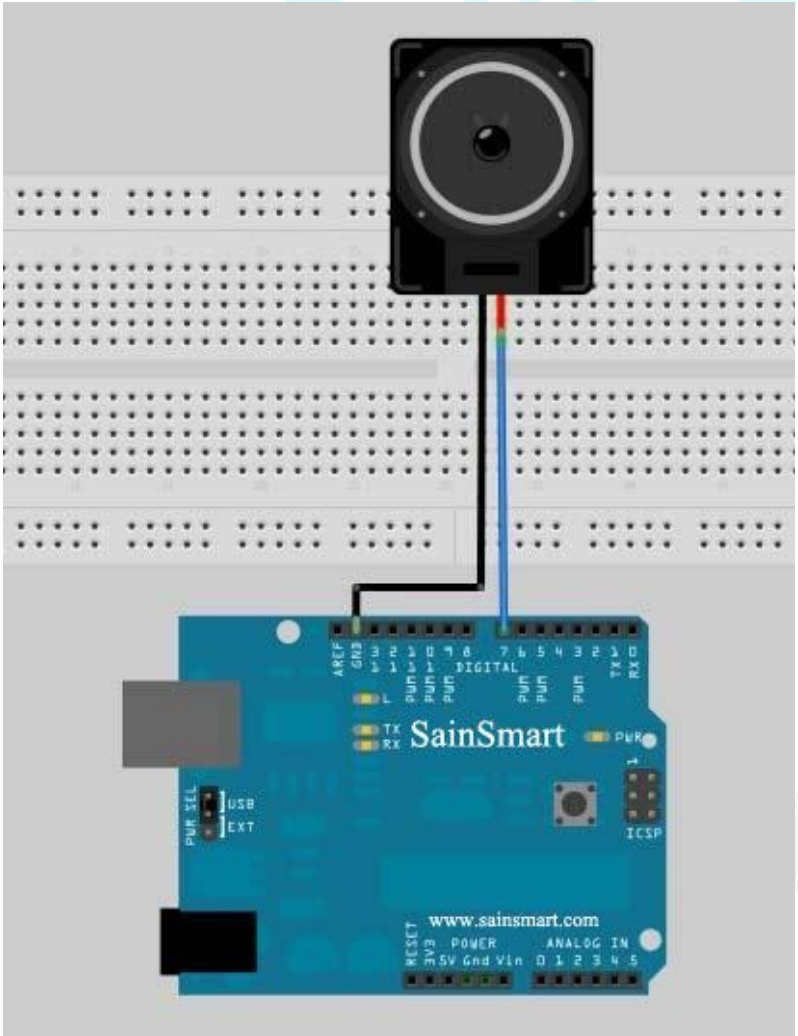


Figure 7.2 Passive buzzer experiment connection.

The siren sketch

Example code from the SainSmart library:

The code for the siren sketch is shown in Figure 7.3.

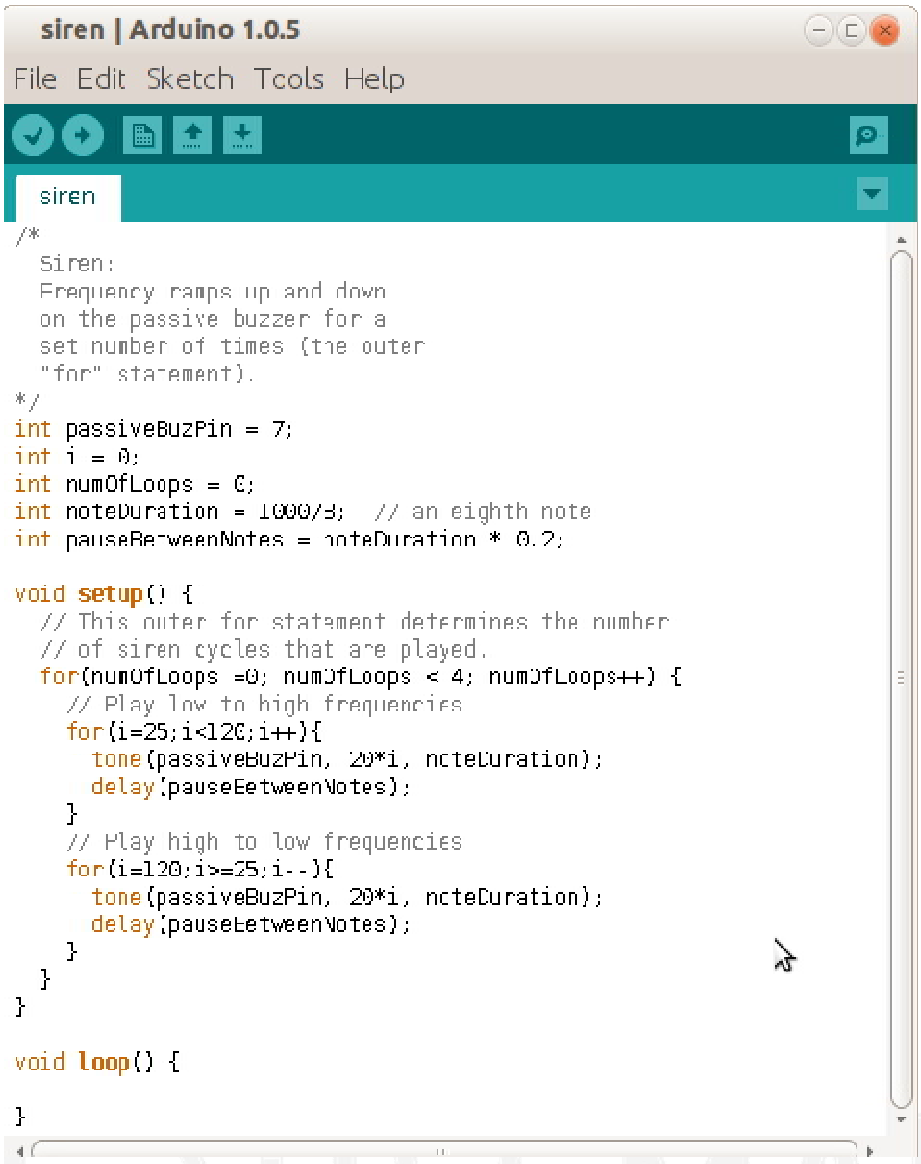


Figure 7.3 Siren sketch.

The siren sketch uses a new control structure, the *for* statement, and the Arduino *tone*() function. Each of these items will be covered in the following sub-sections:

The "for" statement

The *for* statement is used to repeat a block of statements enclosed in curly braces for a set amount of times. An increment counter is used to increment and terminate the loop, see Figure 7.4. The "for" statement is useful for any repetitive operation, and is often used in combination with arrays to operate on collections of data/pins.

There are three parts to the for loop header:

```
for (initialization; condition; increment/decrement) {
//statement(s);
}
```

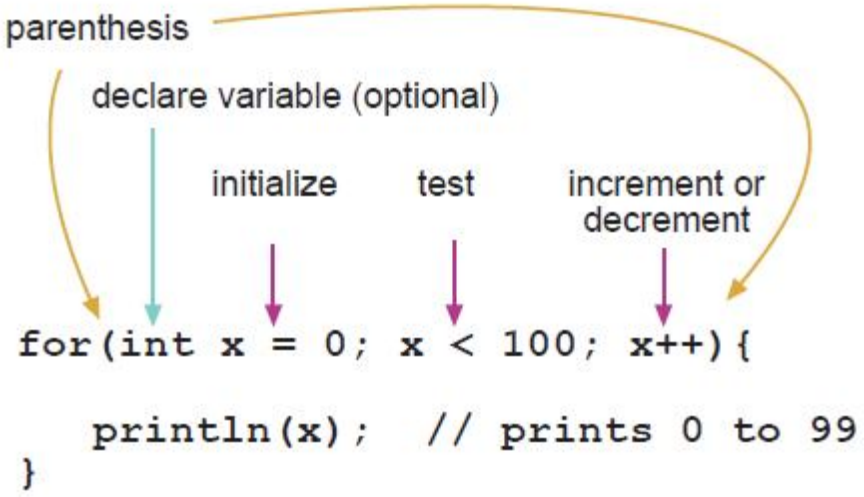


Figure 7.4 The components of a "for" statement.

The initialization happens first and exactly once. Each time through the loop, the condition is tested; if it's true, the statement block, and the increment is executed, then the condition is tested again. When the condition becomes false, the loop ends.

The tone() function

The tone() function is another one of Arduino's built in functions. Its definition and usage can be found at: http://arduino.cc/en/Reference/Tone. The following is from that Webpage:

Description

Generates a square wave of the specified frequency (and 50% duty cycle) on a pin. A duration can be specified, otherwise the wave continues until a call to noTone(). The pin can be connected to a piezo buzzer (passive buzzer) or other speaker to play tones.

Only one tone can be generated at a time. If a tone is already playing on a different pin, the call to tone() will have no effect. If the tone is playing on the same pin, the call will set its frequency.

Use of the tone() function will interfere with PWM output on pins 3 and 11 (on boards other than the Mega).

NOTE: if you want to play different pitches on multiple pins, you need to call noTone() on one pin before calling tone() on the next pin.

Syntax

- tone(pin, frequency)
- tone(pin, frequency, duration)

Parameters

- pin: the pin on which to generate the tone
- frequency: the frequency of the tone in hertz - unsigned int
- duration: the duration of the tone in milliseconds (optional) - unsigned long

Returns

- nothing

Sketch Explained

The siren sketch simply cycles a set number of times (the outer for loop) through an increasing frequency loop and a decreasing frequency loop. The trick is to get a proper balance between the frequency duration and the time between tones. Try different combinations to see how this effects the sound!

The toneMelody sketch

This sketch is less annoying to the ears than the siren sketch. This sketch can be found under: File->Examples->Digital->toneMelody of the Arduino IDE. The source pin selected in this sketch is digital I/O pin 8, so you can either change the code (in the tone and noTone functions), or you will need to change the connection of the plus terminal of the passive buzzer to pin 8. Since a tone duration is used in the tone() function, the use of the noTone() function is not really needed here. Try commenting it out and uploading it to the Uno!

Active buzzer sketch

The active buzzer actually doesn't require the Uno to operate. Simply connecting the positive terminal to +5v and the negative terminal to ground will set it off (hold your ears!). However, to use it as an indicator that a certain event has occurred, having it turned on and off in a certain pattern, these behaviors are easily accomplished through the control of the Uno.

Chapter 8: Flame Sensor

What is a flame sensor?

The flame sensor that comes with the Uno Starter Kits is actually a NPN phototransistor. It looks like an LED, but has a dark casing (see Figure 8.1). The dark casing blocks out most of the visible light and allows for infrared (IR) energy to transmit through. IR waves can't be seen with the naked eye, because their frequency is just below the lower portion of the visible light spectrum, as illustrated in Figure 8.2. But sensors like the phototransistor can pick up their presence. IR waves are normally found emitting from heat sources, such as a flame or even a light bulb (which can get very hot!).

Figure 8.1 Phototransistor

Phototransistor basics

The phototransistor works like a regular transistor, but instead of being turned on and off with an external bias voltage and current source at its base terminal, the base is regulated by the photoelectric effect. A schematic of a common-emitter amplifier configuration is shown in Figure 8.3. This configuration works as an inverter, when used in the high intensity/no intensity IR source mode. Without an IR source, the phototransistor will be off and Vout will be at Vcc (3.3v or 5v). When an IR source is introduced to the circuit, the phototransistor will turn on, allowing for current to flow from Vcc (3.3v or 5v) through the 10 kohm resistor and the phototransitor. Thus the voltage seen at Vout will be a function of the intensity of the IR source, but will be close to GND or 0v when at/near saturation.

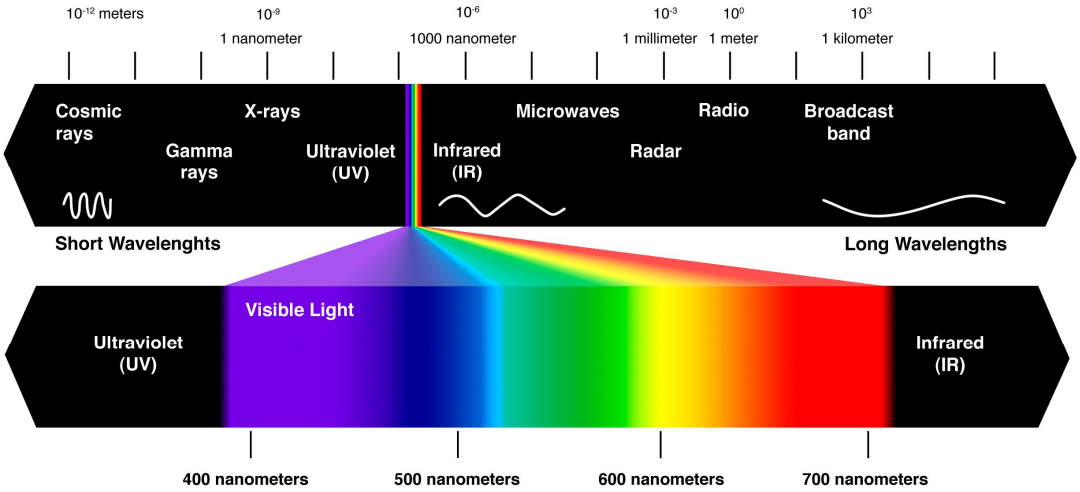


Figure 8.2 Frequency Spectrum
Source: deserthighlandspr.com

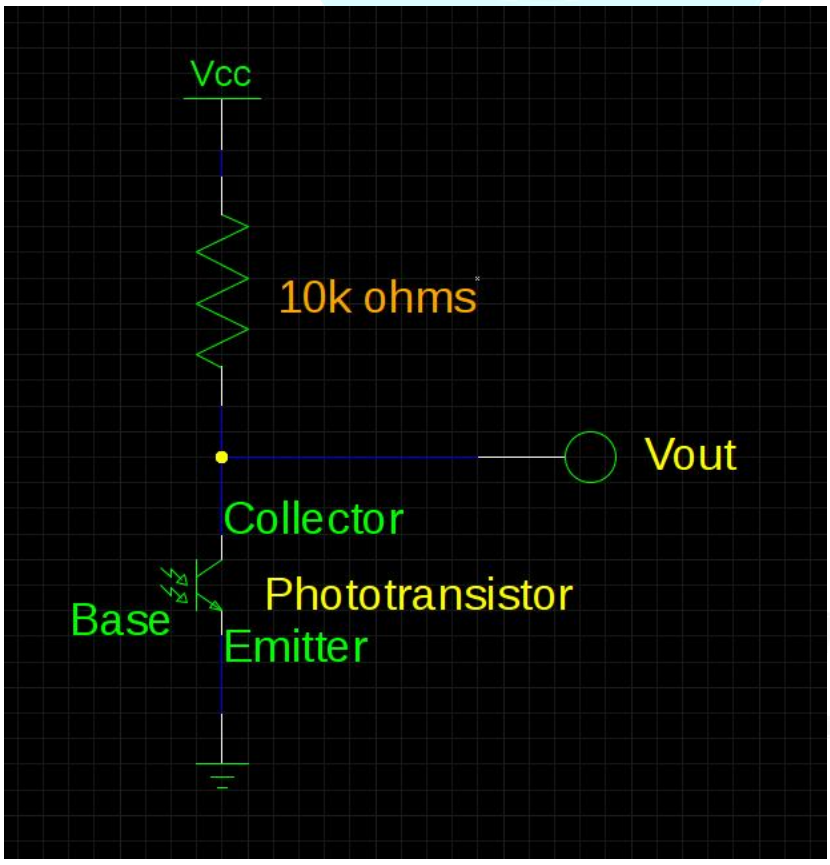


Figure 8.3 Phototransistor common-emitter amplifier circuit.

Note: *The emitter is the longer lead.*

Fade2 sketch circuit

Sketch components:

- 1 x LED (any color, although blue is nice)
- 1 x 1k ohm resistor
- 1 x 10k ohm resistor
- 1 x Phototransistor
- Uno, breadboard, & jumper wires

Connect up the components as shown in Figure 8.4. Take care in connecting the phototransistor and the LED. If the leads on this component are reversed, the circuit will not work.

Note: Don't use the Uno's pin numbers as shown on the outside of the Uno symbol in Figure 8.4. These are the actual pin numbers of the semiconductor package. Instead, use the labels that are inside the box. Example: Digital I/O port 9 is shown as (PB1) **DIGITAL9**. This corresponds to pin 9 on the Uno board/header.

Figure 8.4 Schematic of the fade2 sketch circuit.

Example code from the SainSmart library:

The code for the fade2 sketch is shown in Figure 8.5.

Analog background information for sketch

There are several new programming concepts and Uno ports used in this sketch and circuit. So, first we need cover some background information. Much of this

information comes straight from the Arduino Website (arduino.cc):

What are analog pins?

1. Analog to digital converter

The Atmega328 contains an onboard 6 channel analog-to-digital converter (ADC or A/D). There is actually only one A/D and all of the analog inputs share it through multiplexing. The converter has 10 bit resolution, returning integers from 0 to 1023, where 0 represents 0v and 1023 represents Vcc (3.3v or 5v). While the main function of the analog pins for most Arduino users is to read analog sensors, the analog pins also have all the functionality of general purpose input/output (GPIO) pins (the same as digital pins 0 - 13). So, if a user needs more general purpose input/output pins, and all the analog pins are not in use, the analog pins may be used for GPIO.

2. Pin mapping

The Arduino analog pins use aliases for their numbers. These aliases are A0 thru A5, where "A" stands for analog and the numbers after the "A" corresponds to which analog pin is being used (0 thru 5). For example, to set analog pin 0 to output, and to set it to LOW, the following code needs to be written:

```
pinMode(A0, OUTPUT);
digitalWrite(A0, LOW);
```

3. Pullup resistors

The analog pins also have pullup resistors, which work identically to pullup resistors on the digital pins. They are enabled by issuing a command such as

```
digitalWrite(A0, HIGH);    // set pullup on analog pin 0
```

while the pin is an input.

Be aware however, that turning on a pullup will affect the value reported by analogRead() when using some sensors if done inadvertently. Most users will want to use the pullup resistors only when using an analog pin in its digital mode.

4. Details and Caveats

The analogRead command will not work correctly if a pin has been previously set to an output, so if this is the case, set it back to an input before using analogRead. Similarly if the pin has been set to HIGH as an output, the pullup resistor will be on, after setting it back to an INPUT with pinMode.

The Atmega328 datasheet also cautions against switching digital pins in close temporal proximity to making A/D readings (analogRead) on other analog pins. This can cause electrical noise and introduce jitter in the analog system. It may be desirable, after manipulating analog pins (in digital mode), to add a short delay before using analogRead() to read other analog pins.

analogRead()

Description

Reads the value from the specified analog pin. The Arduino board contains a 6 channel (8 channels on the Mini and Nano, 16 on the Mega), 10-bit analog to digital converter. This means that it will map input voltages between 0 and 5 volts into integer values between 0 and 1023. This yields a resolution between readings of: 5 volts / 1024 units or, .0049 volts (4.9 mV) per unit. The input range and resolution can be changed using analogReference().

It takes about 100 microseconds (0.0001 s) to read an analog input, so the maximum reading rate is about 10,000 times a second.

Syntax

analogRead(pin)

Parameters

pin: the number of the analog input pin to read from (0 to 5 on most boards, 0 to 7 on the Mini and Nano, 0 to 15 on the Mega)

Returns

int (0 to 1023)

Fade2 sketch explained

The fade2 sketch, as shown in Figure 8.5, adjusts the brightness of the LED based on the intensity of an IR source on the phototransistor. As the IR intensity increases, the voltage seen at the analog input pin of the Uno will decrease (as previously explained about the common-emitter amplifier configuration). The sketch used the Arduino map() function (see details below) to reverse scale the A/D value to a PWM value. So, an increase in IR intensity results in a brighter LED. This function can be easily reversed such that an increase in IR intensity results in a dimmer LED. What two ways (one hardware, one software) can this be accomplished?

The map() function

The following is from the Arduino Website (arduino.cc/en/Reference/map):

Usage:
map(value, fromLow, fromHigh, toLow, toHigh)

Description:
Re-maps a number from one range to another. That is, a value of fromLow would get mapped to toLow, a value of fromHigh to toHigh, values in-between to values in-between, etc.

Does not constrain values to within the range, because out-of-range values are sometimes intended and useful. The constrain() function may be used either before or after this function, if limits to the ranges are desired.

Note that the "lower bounds" of either range may be larger or smaller than the "upper bounds" so the map() function may be used to reverse a range of numbers, for example:

y = map(x, 1, 50, 50, 1);

The function also handles negative numbers well, so that this example

y = map(x, 1, 50, 50, -100);

is also valid and works well.

The map() function uses integer math so will not generate fractions, when the math might indicate that it should do so. Fractional remainders are truncated, and are not rounded or averaged.

Parameters

value: the number to map
fromLow: the lower bound of the value's current range
fromHigh: the upper bound of the value's current range
toLow: the lower bound of the value's target range
toHigh: the upper bound of the value's target range

Returns

The mapped value.

Example

```
/* Map an analog value to 8 bits (0 to 255) */
void setup() {}
void loop()
{
  int val = analogRead(0);
  val = map(val, 0, 1023, 0, 255);
  analogWrite(9, val);
}
```

Figure 8.5 Fade2 sketch.

Chapter 9: Tilt Sensor

What is a tilt sensor?

The tilt sensor is a component that can detect the tilting of an object. However it is only the equivalent to a pushbutton activated through a different physical mechanism. This type of sensor is the environmental-friendly version of a mercury-switch. It contains a metallic ball inside that will commute the two pins of the device from on to off and vice versa if the sensor reaches a certain angle.

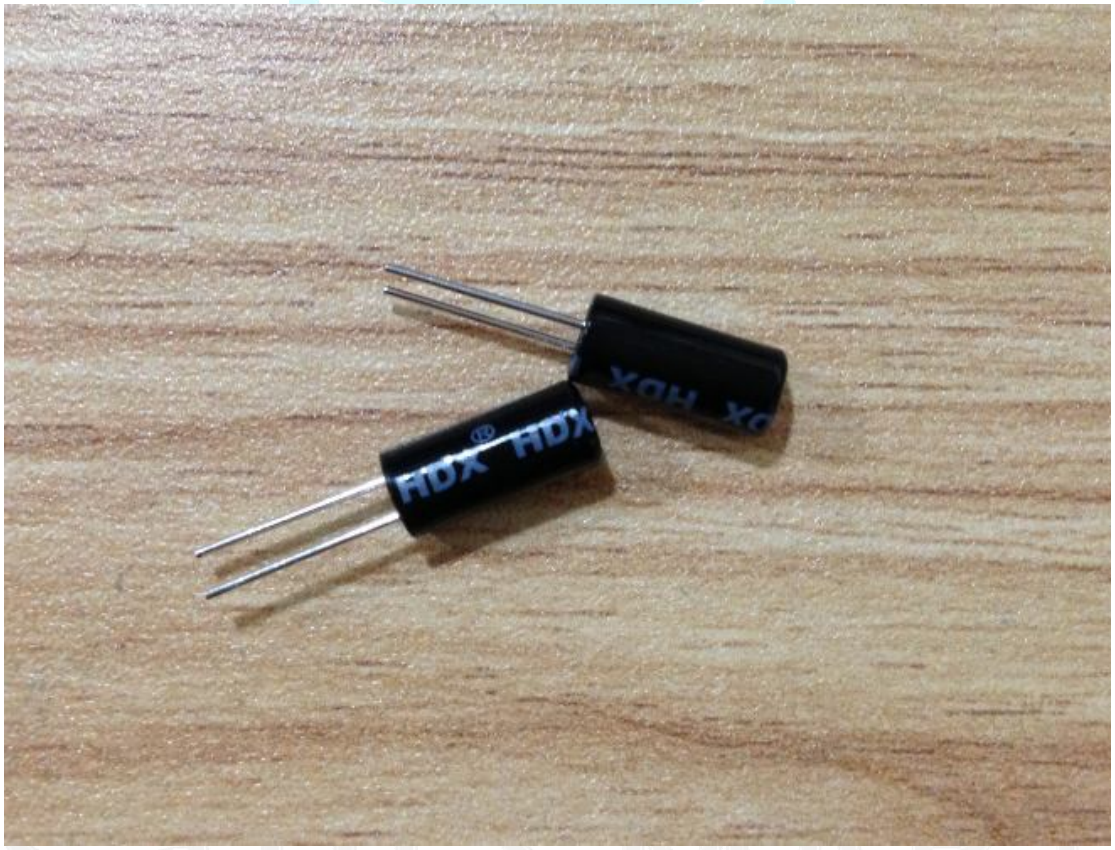


Figure 9.1 Tilt Sensors.

Tilt switch controls LED sketch

Sketch components:

- 1 x Tilt sensor
- 1 x LED (any color)
- Uno, breadboard, & jumper wires

Connect your circuit as shown in Figures 9.2 and 9.3, where one terminal of the tilt

switch connect to analog pin A5.

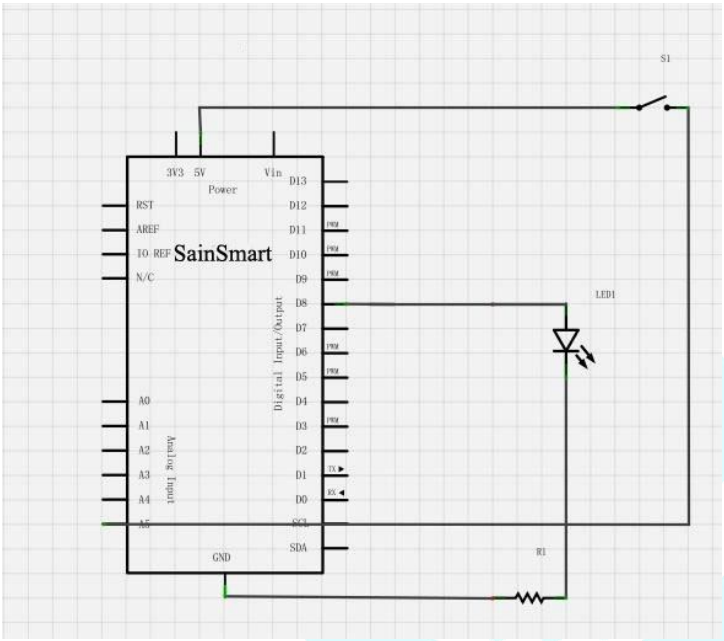


Figure 9.2 Schematic for tilt sensor-LED sketch circuit.

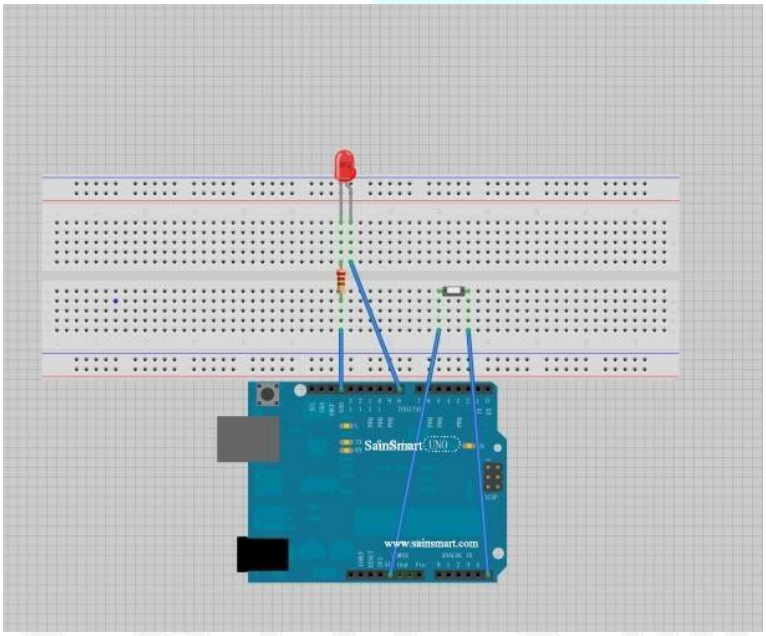


Figure 9.3 Illustration of tilt sensor-LED sketch circuit.

Example code from the SainSmart library:

The code for the tilt sensor-LED sketch is shown in Figure 9.4. It is pretty self-explanatory. The LED will be on while the sensor is mostly in the standing position. Once it is sideways or upside down, the LED will turn off.

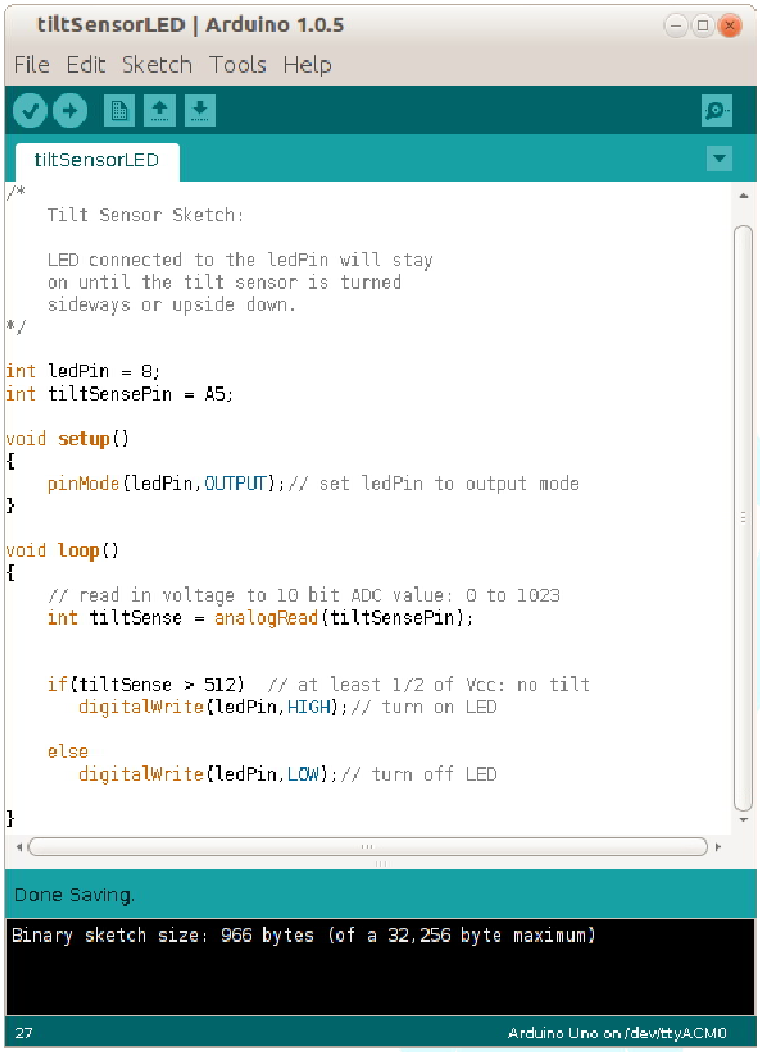


Figure 9.4 Tilt sensor-LED sketch.

Chapter 10: Potentiometer

What is a potentiometer?

A potentiometer is a simple knob that provides a variable resistance, which we can read into the Arduino board as an analog value. In this example, that value controls the rate at which an LED blinks.

We connect three wires to the Arduino board. The first goes to ground from one of the outer pins of the potentiometer. The second goes from 5 volts to the other outer pin of the potentiometer. The third goes from analog input 2 to the middle pin of the potentiometer (see Figure 10.1).

By turning the shaft of the potentiometer, we change the amount of resistance on either side of the wiper which is connected to the center pin of the potentiometer. This changes the relative "closeness" of that pin to 5 volts and ground, giving us a different analog input. When the shaft is turned all the way in one direction, there are 0 volts going to the pin, and we read 0. When the shaft is turned all the way in the other direction, there are 5 volts going to the pin and we read 1023. In between, analogRead() returns a number between 0 and 1023 that is proportional to the amount of voltage being applied to the pin.

Potentiometer sketch

Sketch components:

- 1 x Potentiometer
- Uno, breadboard, & jumper wires

Optional:

- 1 x LED (any color)
- 1 x 1k ohm resistor

The external LED is optional, since digital I/O pin 13 is being used. As discussed in Chapter 2, there is already an LED/series resistor connected to this pin. However, you may want to add an external LED to this port. If so, just follow Figure 4.2 on how to connect up the LED.

Example code from the SainSmart library:

Figure 10.2 shows the code that makes up the potentiometer sketch. The sketch is pretty straight forward. As described above, it performs an analogRead to obtain the A/D value from the analog port that the potentiometer. This value is used for two things: 1) it is used to set the LED on and off time intervals (0ms to 1023ms), and 2) the value is sent back to the computer and will be displayed in the monitor window.

Figure 10.3 shows an example output that is written to the monitor window.

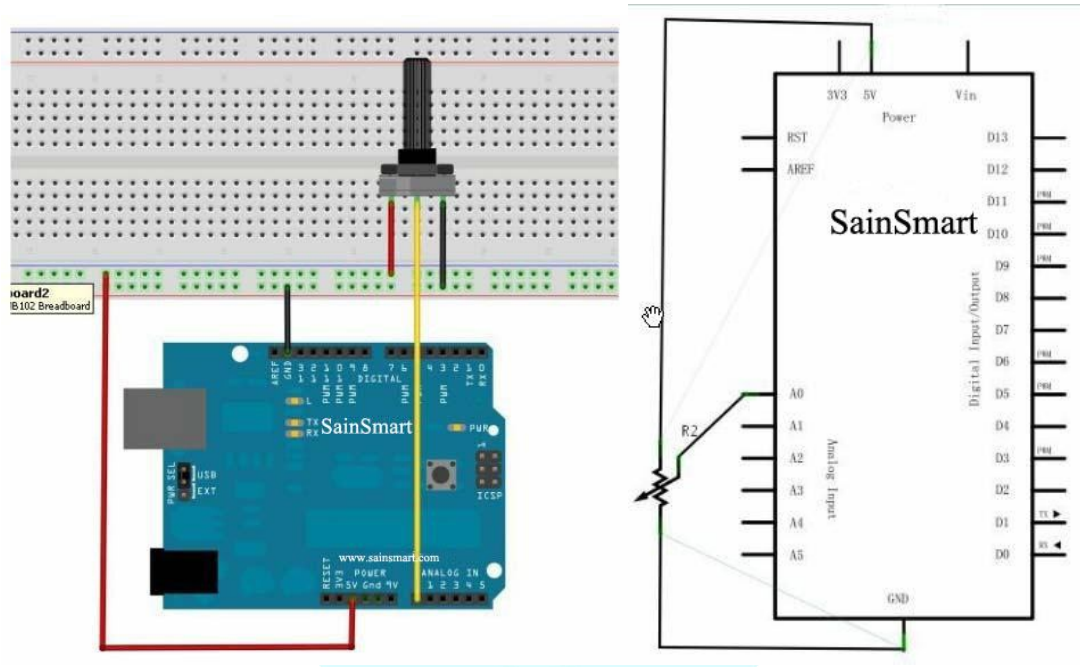


Figure 10.1 Potentiometer sketch circuit.

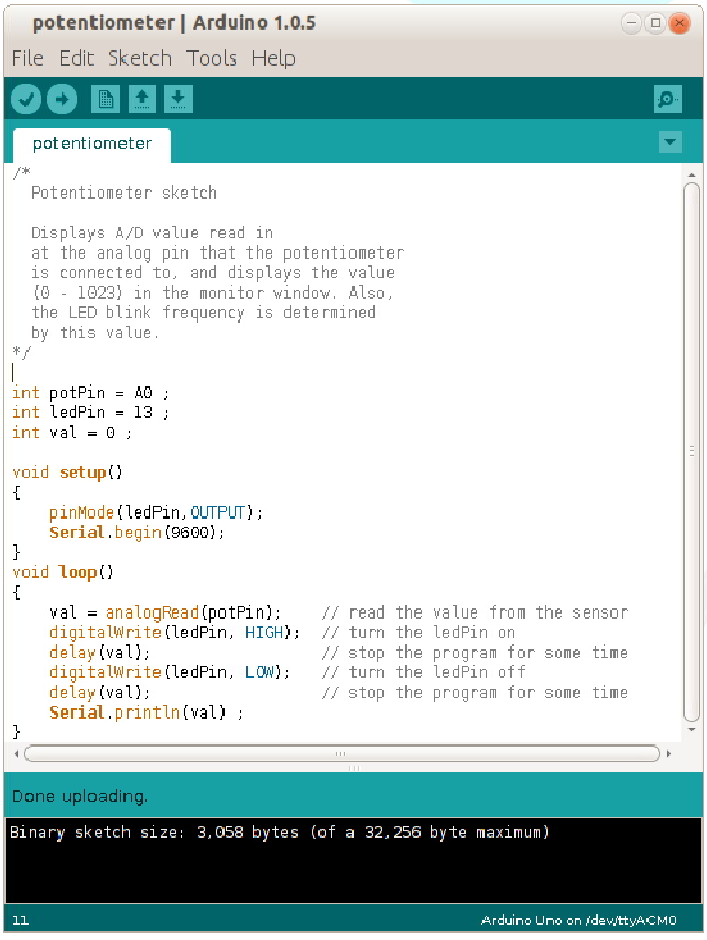


Figure 10.2 Potentiometer sketch.

Figure 10.3 Monitor display of potentiometer sketch output.

Chapter 11: Photoresistor

What is a photoresistor?

A photoresistor is a device whose resistance is a function of the light intensity that is incident upon it. Increasing the intensity of light that is incident upon the photoresistor's "face", decreased the resistance seen across its terminals.

Figure 11.1 Photoresistor

Photoresistor low light level alarm sketch

Sketch components

- 1 x Photoresistor
- 1 x Active Buzzer
- 1 x LED (red is cool for this)
- 1 x 10k ohm resistor
- 1 x 220 ohm resistor
- Uno, breadboard, & jumper wires

Description

Connect your circuit as shown in the schematic in Figure 11.2. Previous circuits use a 1k ohm resistor in series with the LED. This is to help prolong its life by cutting down on the input current. However, the red LED is a little harder to see, so a 220 ohm resistor is suggested and the life of the LED shouldn't be compromised due to the relatively long off times.

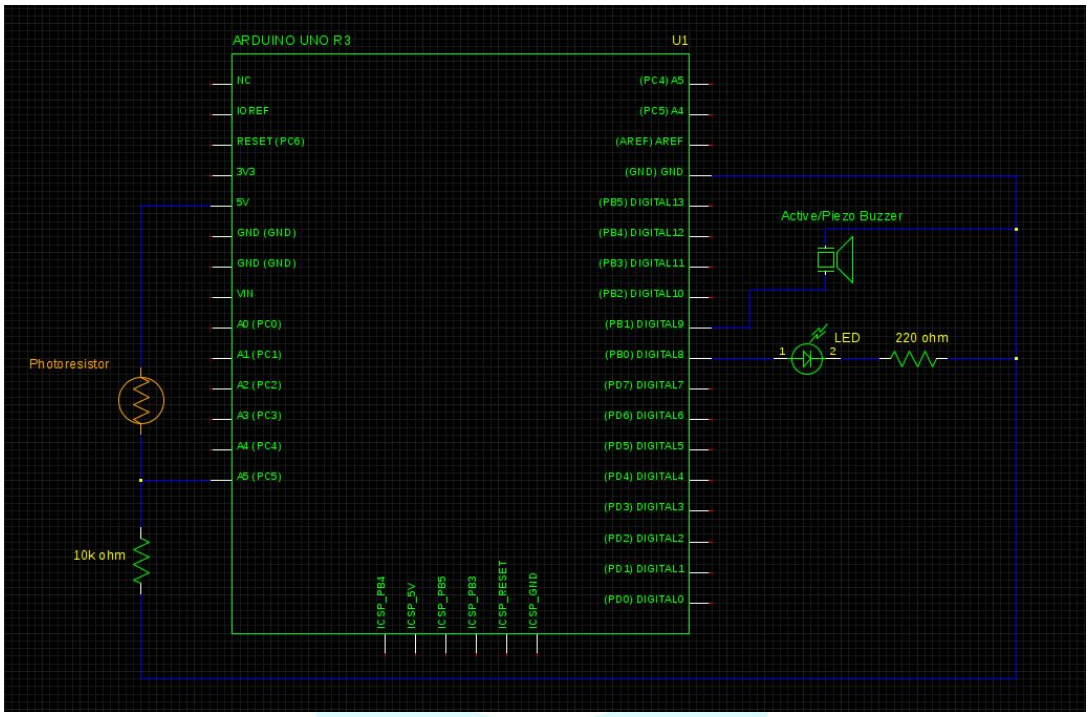


Figure 11.2 Photoresistor sketch circuit.

Example code from the SainSmart library:

Figure 11.3 shows a copy of the photoresistor low light level sketch. The header of the sketch gives a brief description of how this sketch works. The following is a more detailed version of this description plus more information as to how this circuit works:

With low light levels, or no light, the resistance of the photoresistor increases into the 10's of thousands of ohms. This causes the voltage seen between the photoresistor and the 10k ohm resistor, voltage divider circuit, to decrease. An "alarm" will sound by turning on and off the buzzer if the voltage divider voltage, seen at the analog port, dips below a threshold value set in the sketch. The lower the light value below the threshold, the faster the on/off rate of the LED and the buzzer. You can experiment with this by having a light source over the photoresistor and then placing your hand over the photoresistor.

Figure 11.3 Photoresistor Low Light Level Alarm sketch.

Chapter 12: LM35 Temperature Sensor

Temperature sensor

The following detailed description of the LM35 comes from the Texas Instrument datasheet on this part (www.ti.com/lit/ds/symlink/lm35.pdf):

> The LM35 series are precision integrated-circuit temperature sensors, with an output voltage linearly proportional to the Centigrade temperature. Thus the LM35 has an advantage over linear temperature sensors calibrated in ° Kelvin, as the user is not required to subtract a large constant voltage from the output to obtain convenient Centigrade scaling. The LM35 does not require any external calibration or trimming to provide typical accuracies of ±1⁄4°C at room temperature and ±3⁄4°C over a full −55°C to +150°C temperature range. Low cost is assured by trimming and calibration at the wafer level. The low output impedance, linear output, and precise inherent calibration of the LM35 make interfacing to readout or control circuitry especially easy. The device is used with single power supplies, or with plus and minus supplies. As the LM35 draws only 60 µA from the supply, it has very low self-heating of less than 0.1°C in still air. The LM35 is rated to operate over a −55°C to +150°C temperature range.

Figure 12.1 shows a picture of the LM35 precision centigrade temperature sensor that comes with the Uno Starter Kits.

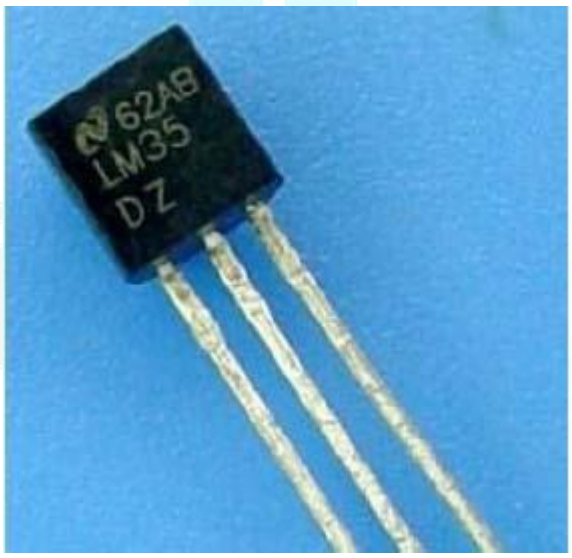


Figure 12.1 LM35 Precision Centigrade Temperature Sensor.

Working principle

LM35 temperature sensor has a linear relationship between the Celsius temperature scale and its output voltage. The relationship is: 0 ℃ = 0V output, and for every 1℃ increase in temperature = 10mV in output voltage. This relationship between the LM35 output voltage, Vout, and the temperature (t) surrounding the LM35, in ℃, is given by equation 12.1.

$$V_{out\_LM35}(t) = 10mV/℃ \times t\ (℃) \qquad \text{(Eqn: 12.1)}$$

The pinout for the LM35 is shown in Figure 12.2. The sensor works at +Vs of 4 to 30 Volts, so the Uno needs to be set at the 5V setting when using this part.

Figure 12.2 LM35 Pinout.

Temperature display sketch

Sketch components

- 1 x LM35 temperature sensor module
- Uno, breadboard, & jumper wires

Connection

Figure 12.3 illustrates the connection for the LM35 temperature sensor sketch. The N shown on the LM35 in this figure represents the flat face side.

Sketch description

The code for this sketch can be uploaded to your Uno from the SainSmart library. This sketch uses equation 12.1 to calculate the temperature near the LM35 by sampling its output voltage, which is a connected to one of the Uno's analog input

pins.
According to this principle procedures in real time reading out the analog voltage value of 0, since the analog port reads out a voltage value of 0 to 1023, i.e. 0V corresponding 0,5 V corresponds to 1023.
Application, we only need to LM35 module, analog interface, the read analog value is converted to the actual temperature.

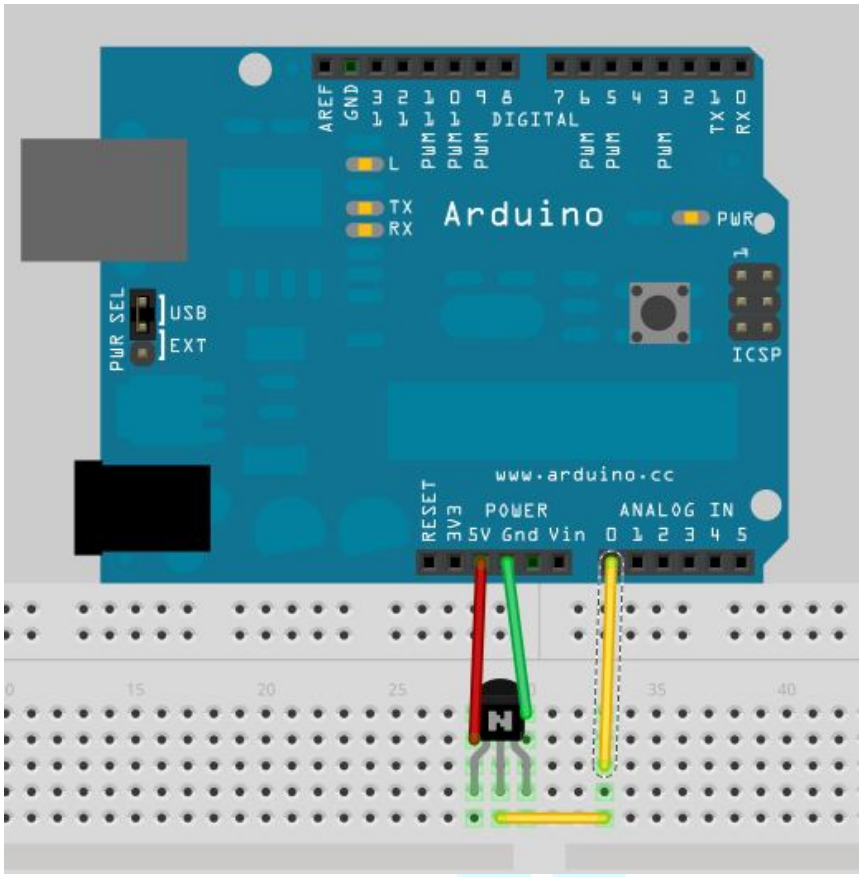


Figure 12.3 Schematic for the LM35 temperature sensor sketch.

Sketch description

Download the program to the Uno from the SainSmart library (Figure 12.4). Then open the Serial Monitor. You will see the current ambient temperature in both Celsius and Fahrenheit displayed (Figure 12.5).

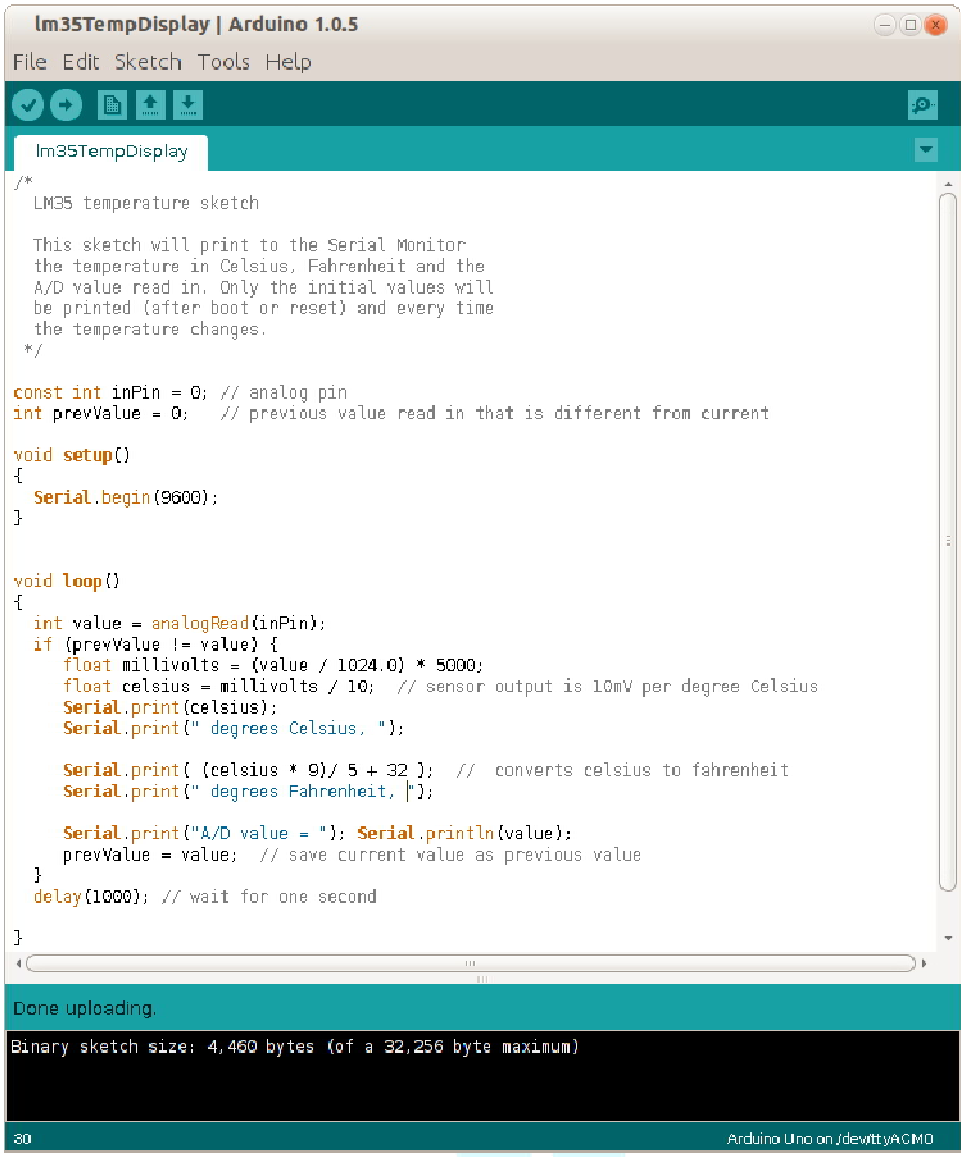


Figure 12.4 LM35 Temperature Display sketch.

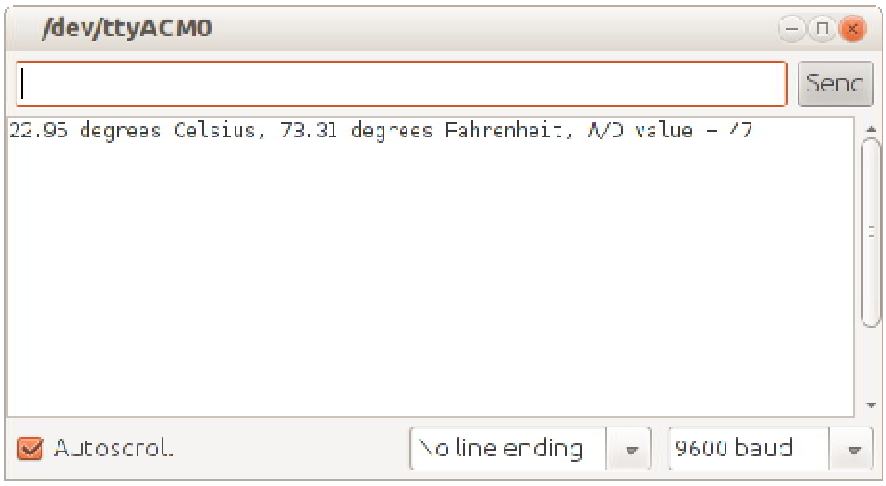


Figure 12.5 Serial Monitor display of running sketch.

Chapter 13: 7-Segment Display

The 7-segment display is, as its name implies, made up of seven LED segments, plus a decimal point (DP) LED, as shown in Figure 13.1. The different combinations of on and off of the seven segments allows for the representation of the decimal numbers 0 thru 9, and 0 thru F for hexadecimal values. The segment's location is described by the letters a thru g in technical literature and pinouts. The letter is merely a label given to a particular segment, as shown in Figure 13.2. The seven segment displays with the number: **5101AS** printed on the side is a **common cathode** type display.

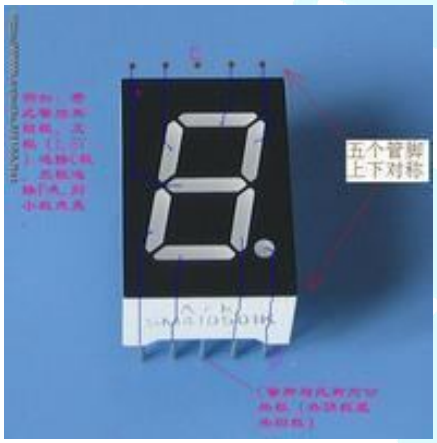


Figure 13.1 7-segment LED

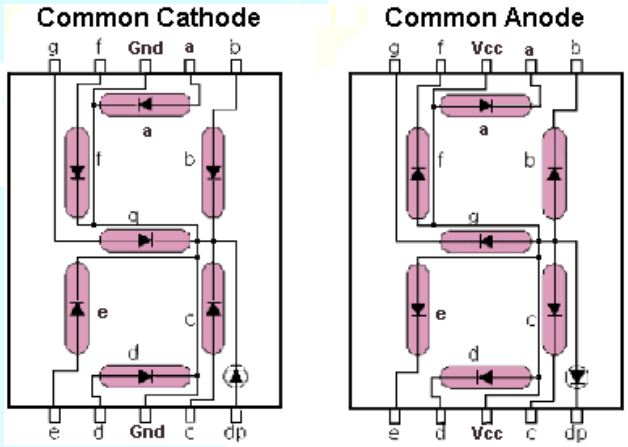


Figure 13.2 Common Cathode and Anode pinouts. (Source: www.thelearningpit.com)

Counting Numbers Sketch

Sketch components:

- 1 x 7-segment display
- 7 x 220 Ω resistors
- Uno, breadboard, & jumper wires

Circuit Description

Connect your circuit as shown in Figure 13.2. It takes some creativity to make the connections look "pretty", so if you would like to add extra jumpers to do so, please go ahead. Just make sure that the connection listed in Table 13.1 are made one way or another.

7-Segment Pin	Connection Medium	Uno Digital I/O Pin
Gnd	wire	Gnd
a	220 Ω resistor	2
b	220 Ω resistor	3
c	220 Ω resistor	4
d	220 Ω resistor	6
e	220 Ω resistor	7
f	220 Ω resistor	8
g	220 Ω resistor	9
DP	No Connect (NC)	NC

Table 13.1 Sketch circuit connections

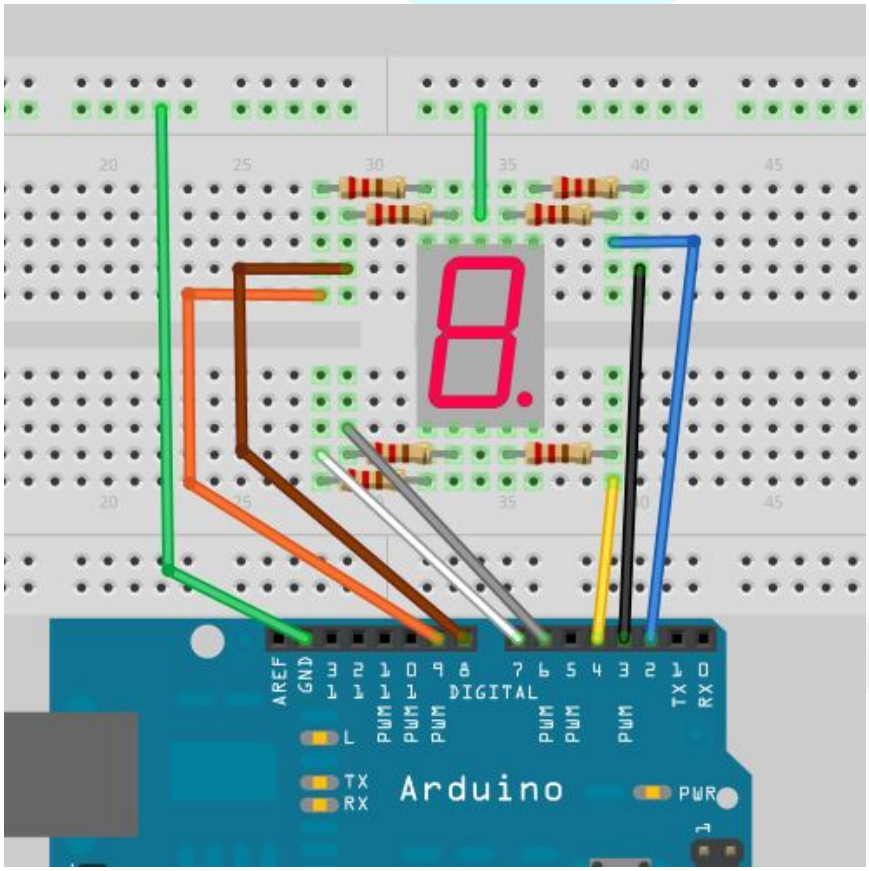


Figure 13.2 Seven segment display connected to the Uno.

The code for this sketch can be found in the SainSmart library.

Chapter 14: Multi-digit, 7-Segment Display

The multi-digit, 7-segment display is an integration of several 7-segment displays into a single package, as illustrated in Figure 14.1.

How does the multi-digit, 7 segment display work?

There are two types of multiplexing that takes place to allow for the display of multiple numbers/digits on this display. The first type is built into the device itself. It is the sharing of the segment pins/traces by all of the digits, as shown in Figure 14.2. There is a single pin that connects all of the "a" segments, a single pin that connects all of the "b" segments, etc. This hardwire multiplexing is possible because only one digit is enabled at a given time (sourcing the common anode or sinking the common cathode). If multiple digits are enabled at the same time, then they will all display the same number or character.

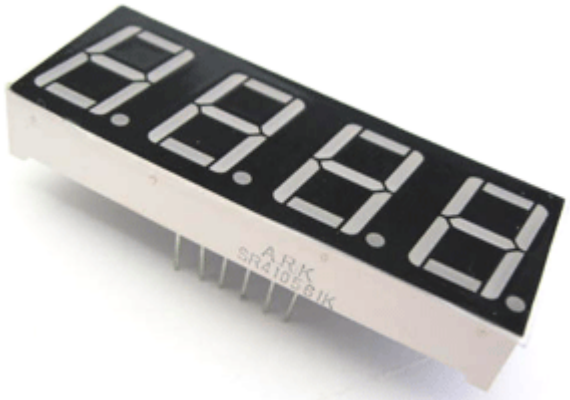


Figure 14.1 A 4-digit, 7-segment display.

That brings us to the second type of multiplexing: time division multiplexing. Each digit takes a turn being enabled to display a particular digit. When this is done fast enough, the eye/brain is not able to tell the difference. It appears that they are all lit at the same time, when in fact they aren't.

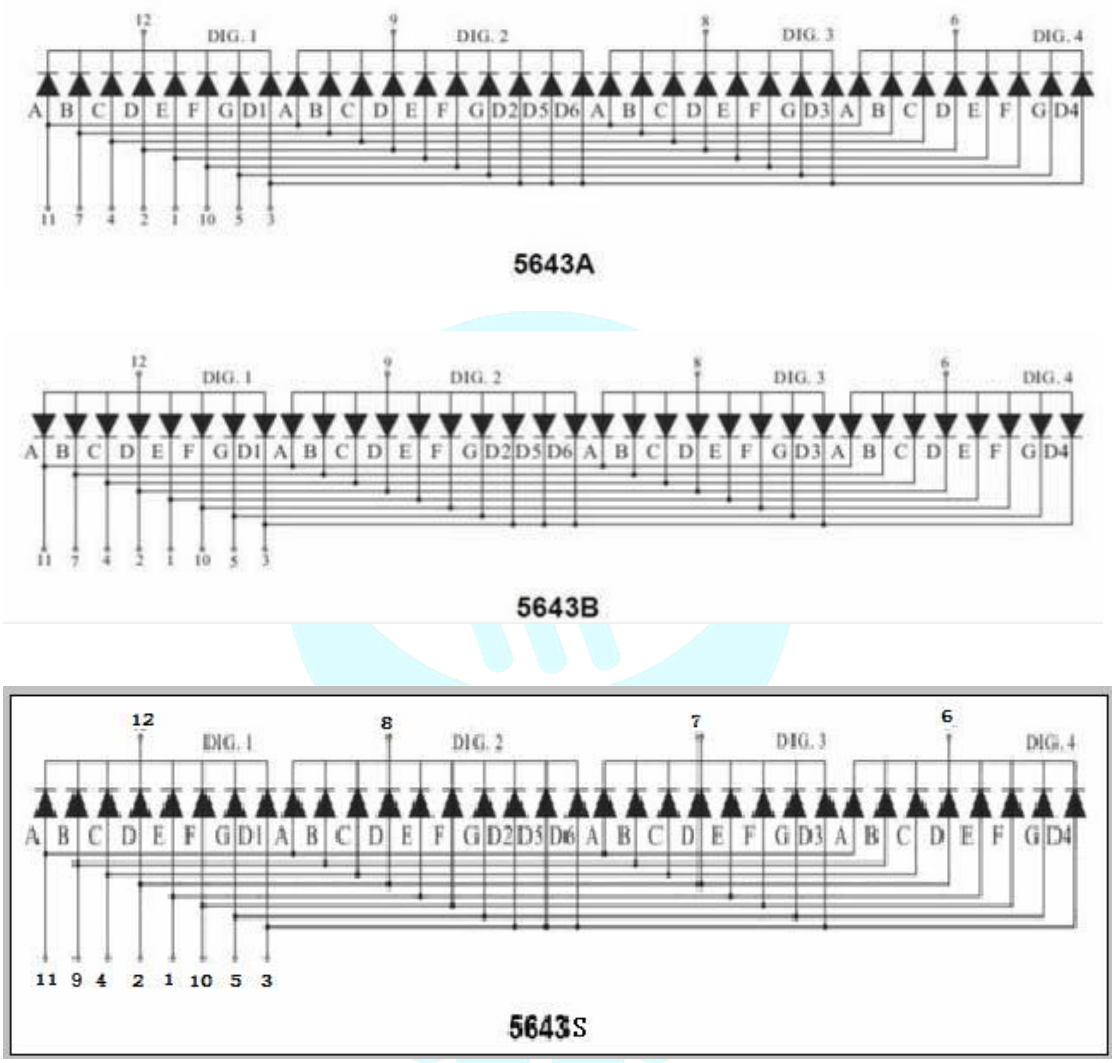


Figure 14.2 Pinouts for various types of multi-digit, 7-segment displays.

Connection

There are a total 12 pins in one 4-digit display. The decimal point is at the bottom, when viewed straight on. As you can see from Figure 14.3, Pin 1 is in the lower left bottom. The other pins' sequences are rotated counterclockwise. Upper left corner is the largest, 12th pin.

To figure out which type of display you have (as shown in Figure 14.2), you should test first for common anode or common cathode. If you determine that you have a common cathode (Gnd), then you need to check to see if pin 7 or pin 9 is connected to segment b (packages 5643A/S in Figure 14.2). Remember to use a series resistor when testing out the segment LEDs. The packages labeled: *LD-5461AS* and *5641AS* on their sides, both have the same polarity and pinout as the package *5643A* in Figure 14.2.

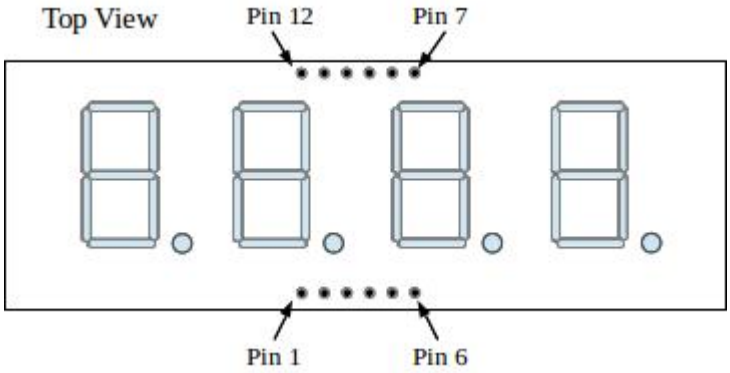


Figure 14.3 Pin numbers for the 4-digit, 7-segment display.

Stopwatch sketch

Sketch components

- 1 x 4-digit, 7-segment display module
- 8 x 220 Ω resistors
- 1 x push-button
- Uno, breadboard, & jumper wires

Connection

The circuit connection for the stopwatch sketch is illustrated in Figure 14.4. The sketch and the wiring diagram are written with respect to the 5643A type package.

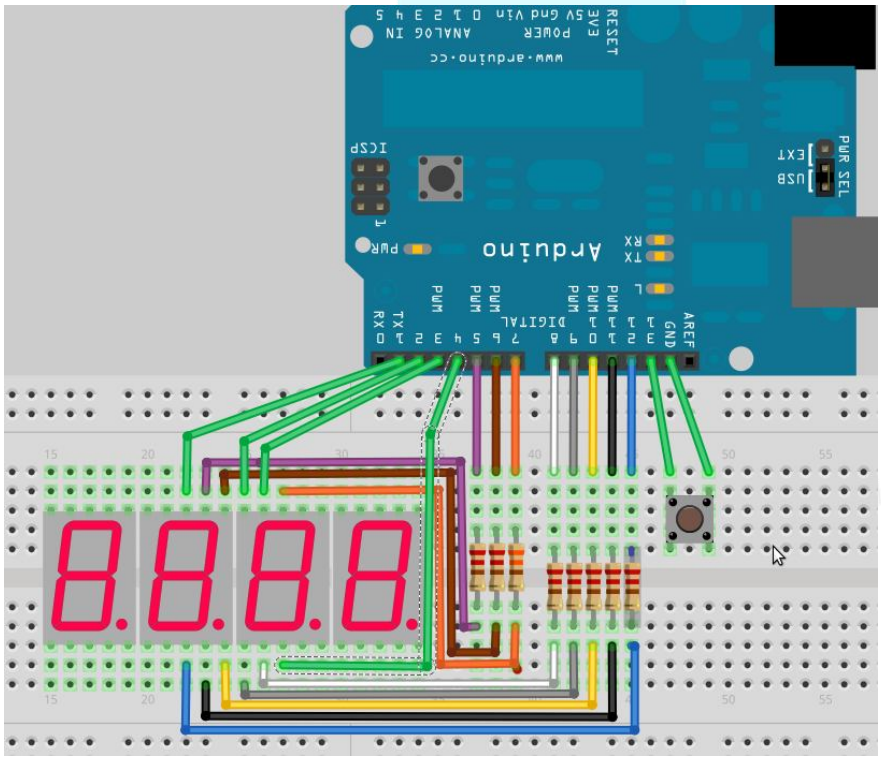


Figure 14.4 Stopwatch circuit for the 5643A type package.

Sketch description

The code for this sketch can be uploaded to your Uno from the SainSmart library. Figure 14.5 (a thru e) shows a snippet of this code, from the beginning, up to the end of the pickNumber() function. The remainder of the code is self-explanatory and is similar to that found in the 7-segment chapter's sketch.

As can be seen from Figure 14.5a, the Uno's digital I/O ports 1 thru 13 are used. Pin 13 of the Uno is set as an input port and the internal pull-up resistor is enabled. This allows for the program to change states when this pin is momentarily pulled low, as in this case, through a pushbutton. The pushbutton can be seen in Figure 14.6, on the right side of the breadboard.

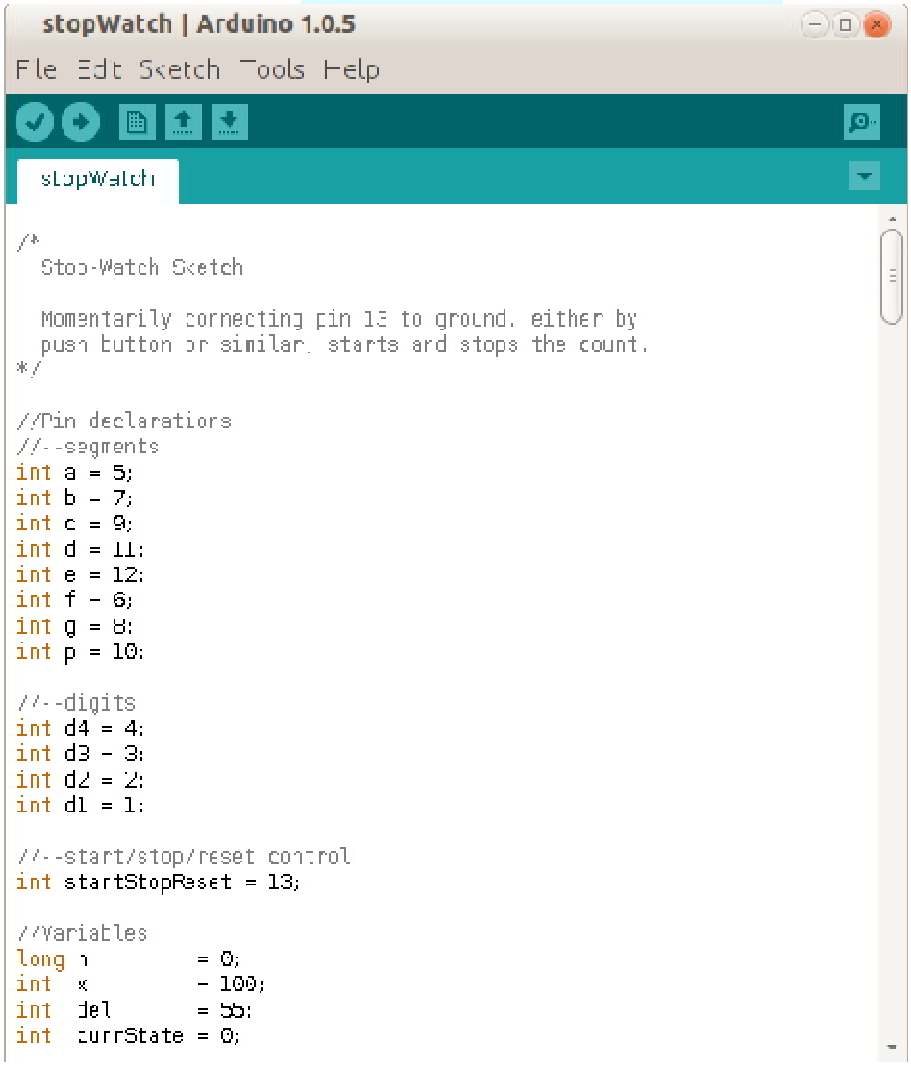


Figure 14.5a Stopwatch sketch code.

```
void setup()
{
  pinMode(d1, OUTPUT);
  pinMode(d2, OUTPUT);
  pinMode(d3, OUTPUT);
  pinMode(d4, OUTPUT);
  pinMode(a, OUTPUT);
  pinMode(b, OUTPUT);
  pinMode(c, OUTPUT);
  pinMode(d, OUTPUT);
  pinMode(e, OUTPUT);
  pinMode(f, OUTPUT);
  pinMode(g, OUTPUT);
  pinMode(p, OUTPUT);
  pinMode(startStopReset, INPUT);
  digitalWrite(startStopReset, HIGH); //set pull-up resistor on
}

void loop()
{
  int swButtonState = digitalRead(startStopReset);

  if (swButtonState == LOW) {
     currState++;
     while (digitalRead(startStopReset) == LOW) {}
  }
```

Figure 14.5b Stopwatch sketch code.

```
  clearLEDs();
  pickDigit(1);
  pickNumber((n/x/1000)%10);
  delayMicroseconds(del);

  clearLEDs();
  pickDigit(2);
  pickNumber((n/x/100)%10);
  delayMicroseconds(del);

  clearLEDs();
  pickDigit(3);
  dispDec(3);
  pickNumber((n/x/10)%10);
  delayMicroseconds(del);

  clearLEDs();
  pickDigit(4);
  pickNumber(n/x%10);
  delayMicroseconds(del);

  if ((currState%3) == 0)      //reset state
     n = 0;
  else if ((currState%3) == 1) //start state
     n++;
  else // stop state
  {}

}
```

Figure 14.5c Stopwatch sketch code.

```
  switch(x)
  {
  case 1:
    digitalWrite(d1, LOW); //100's of seconds
    break;
  case 2:
    digitalWrite(d2, LOW); //10's of seconds
    break;
  case 3:
    digitalWrite(d3, LOW); //unit seconds
    digitalWrite(p, HIGH); //turn on decimal led for digit 3
    break;
  default:
    digitalWrite(d4, LOW); //tenth of a second
    break;
  }
}
```

Figure 14.5d Stopwatch sketch code.

```
void pickNumber(int x)
{
  switch(x)
  {
  default:
    zero();
    break;
  case 1:
    one();
    break;
  case 2:
    two();
    break;
  case 3:
    three();
    break;
  case 4:
    four();
    break;
  case 5:
    five();
    break;
  case 6:
    six();
    break;
  case 7:
    seven();
    break;
  case 8:
    eight();
    break;
  case 9:
    nine();
    break;
  }
}
```

Figure 14.5e Stopwatch sketch code.

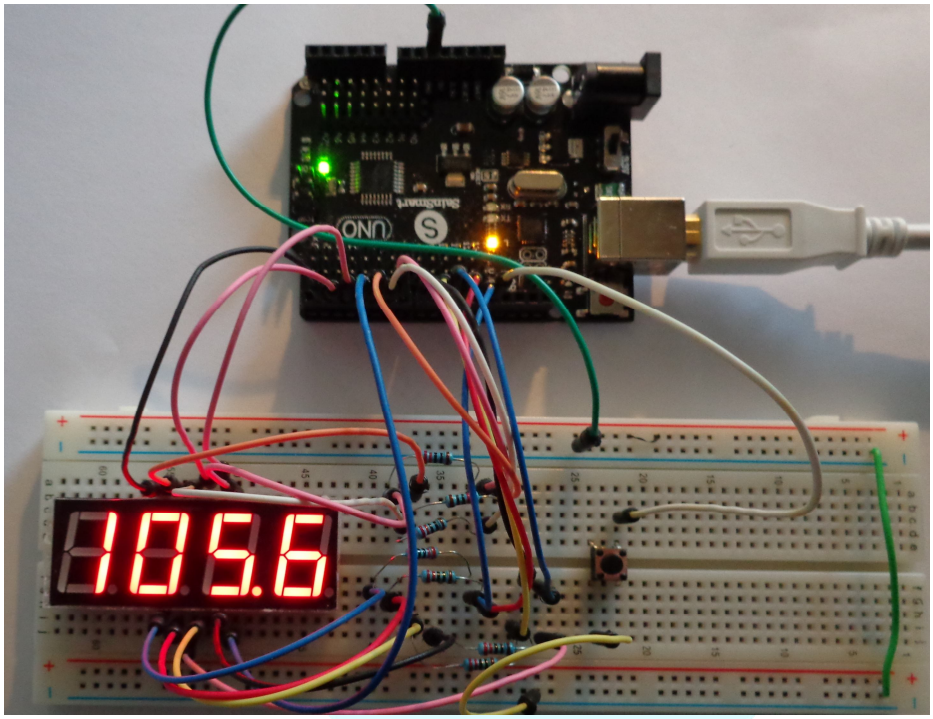


Figure 14.6 Actual circuit in the "Stop" state.

The variables ***currState*** and ***n*** are used as counters. The value of currState modulus 3 determines which state the circuit is in: 0 = RESET, 1 = START (count), and 2 = STOP (hold/display last value counted). The variable currState is incremented once when pin 13 of the Uno goes low. Once it goes high again, the circuit finishes displaying the current value of ***n*** and then either sets it to 0 (reset state), increments it (start state), or leaves it as is (stop state).

Chapter 15: 74HC595

What is the 74HC595?

The 74HC595 is composed of 8 x 1-bit shift registers and 8 x 1-bit storage registers with 3-state output. Data is loaded in serially, and it can be output in serial or parallel. Using this module as a serial to parallel converter reduces the number of I/O ports required to drive 8 bits of digital output, or multiples of 8 bits if more than one 74HC595 module are serially chained together.

Figure 15.1 shows the pin assignments for this chip and Table 15.1 describes the pins' functionality.

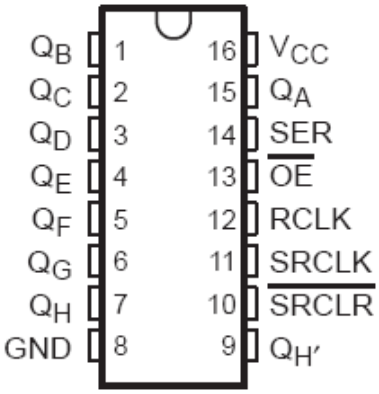


Figure 15.1 Pin assignments for the 74HC595.

Pin #	Pin Name	I/O	Function/Description
15, 1-7	Q_A thru Q_H	O	Parallel output data.
8	GND	I	IC ground in.
9	$Q_{H'}$	O	Serial bit out.
10	!SRCLR	I	Master clear for serial registers (active low).
11	SRCLK	I	Master clock for serial registers.
12	RCLK	I	Master clock for output registers.
13	!OE	I	Master control for output buffers enable (active low).
14	SER	I	Serial data bit in.
16	Vcc	I	IC positive power in.

Table 15.1 Pin descriptions for the 74HC595.

A great document for additional details and timing specifications can be found in the Texas Instruments (TI) datasheet for the 74HC595. The logic diagram of this chip, as

copied from the TI datasheet, is shown in Figure 15.2 and its function table is shown in Figure 15.3.

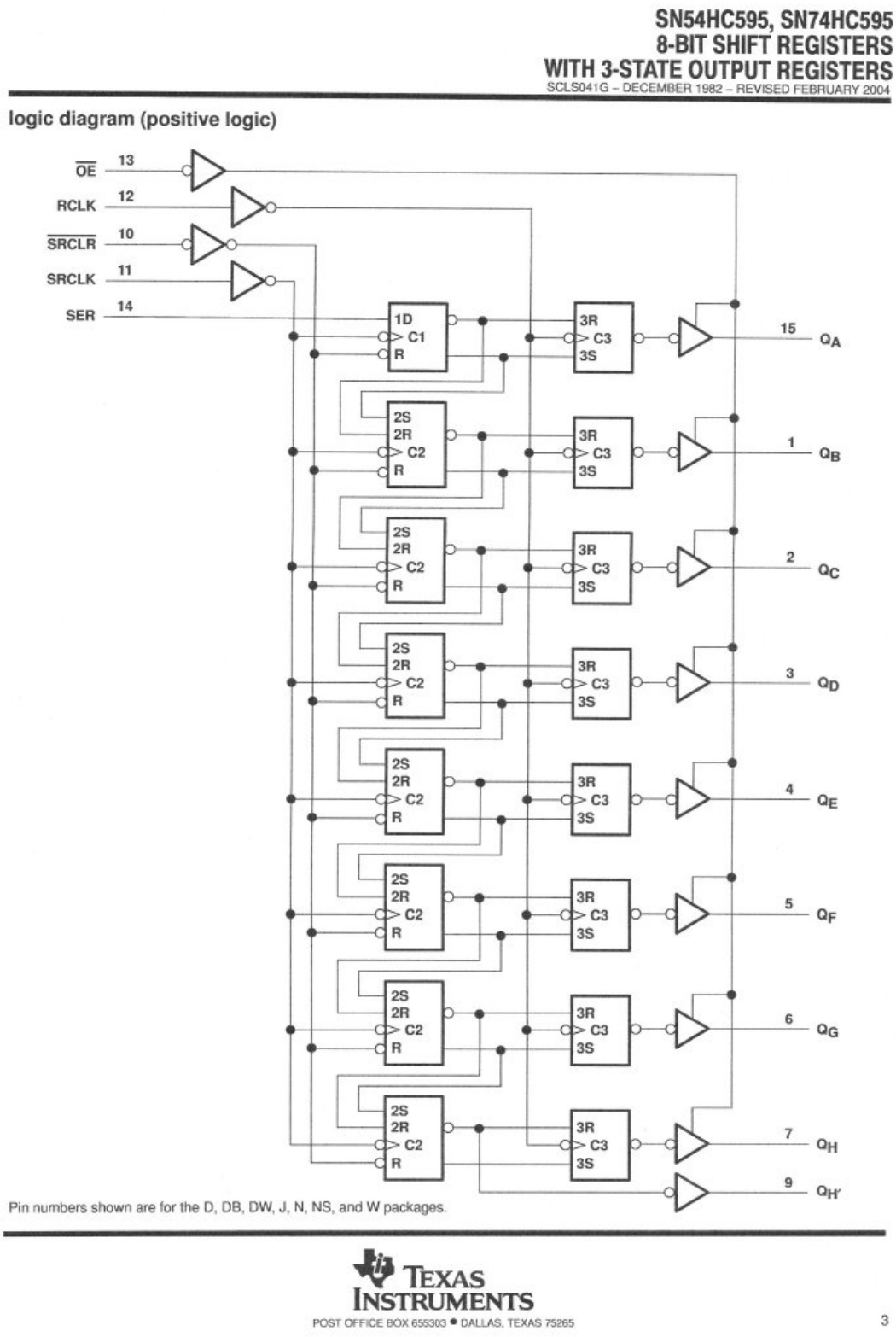


Figure 15.2 Logic diagram of the 74HC595.
(Source: www.ti.com/lit/ds/symlink/sn74hc595.pdf)

FUNCTION TABLE

INPUTS					FUNCTION
SER	SRCLK	$\overline{\text{SRCLR}}$	RCLK	$\overline{\text{OE}}$	
X	X	X	X	H	Outputs Q_A–Q_H are disabled.
X	X	X	X	L	Outputs Q_A–Q_H are enabled.
X	X	L	X	X	Shift register is cleared.
L	↑	H	X	X	First stage of the shift register goes low. Other stages store the data of previous stage, respectively.
H	↑	H	X	X	First stage of the shift register goes high. Other stages store the data of previous stage, respectively.
X	X	X	↑	X	Shift-register data is stored in the storage register.

Figure 15.3 Input/output function table for the 74HC595.
(Source: www.ti.com/lit/ds/symlink/sn74hc595.pdf)

LED light pattern sketch

Sketch components

- 1 x 74HC595 module
- 4 x red LEDs
- 4 x blue LEDs
- 8 x 220 Ω resistors
- Uno, breadboard, & jumper wires

Connections

The circuit connection for this sketch is illustrated in Figure 15.4. As can be seen from this figure, of the seven input pins, four of them are hard wired as follows:

Vcc (pin16) to 5v,
GND (pin 8) to ground,
!OE (pin 13) to ground (enabled),
!SRCLR (pin 10) to 5v (disabled).

This leaves three input pins to be controlled by the Uno: RCLK, SRCLK, and SER. They are wired as follows:

SER (pin 14) to Uno, pin 2,
RCLK (pin 12) to Uno, pin 4,
SRCLK (pin 11) to Uno, pin 5.

The alternating color diodes with series 220 ohm resisters are connected between outputs Q_A thru Q_H and ground.

Sketch description

The code for this sketch can be uploaded to your Uno from the SainSmart library. A copy of this code is show in Figure 15.5.

An array of 8-bit patterns named bits[] is used to store the different LED patterns that will be displayed. Each 8-bit pattern or "word" will be stepped through a loop to

check the value of each bit in that word, then output it to the serial data registers via the latchPin signal (SRCLK) and dataPin (SER). After the 8 bits have been shifted into the 74HC595 from the Uno, the clockPin signal (RCLK) is strobed to parallel load the 8 bit value from the shift registers into the output registers. Since the output enable (!OE) buffers are already enabled, the data pattern shows up at the Q_A thru Q_H outputs.

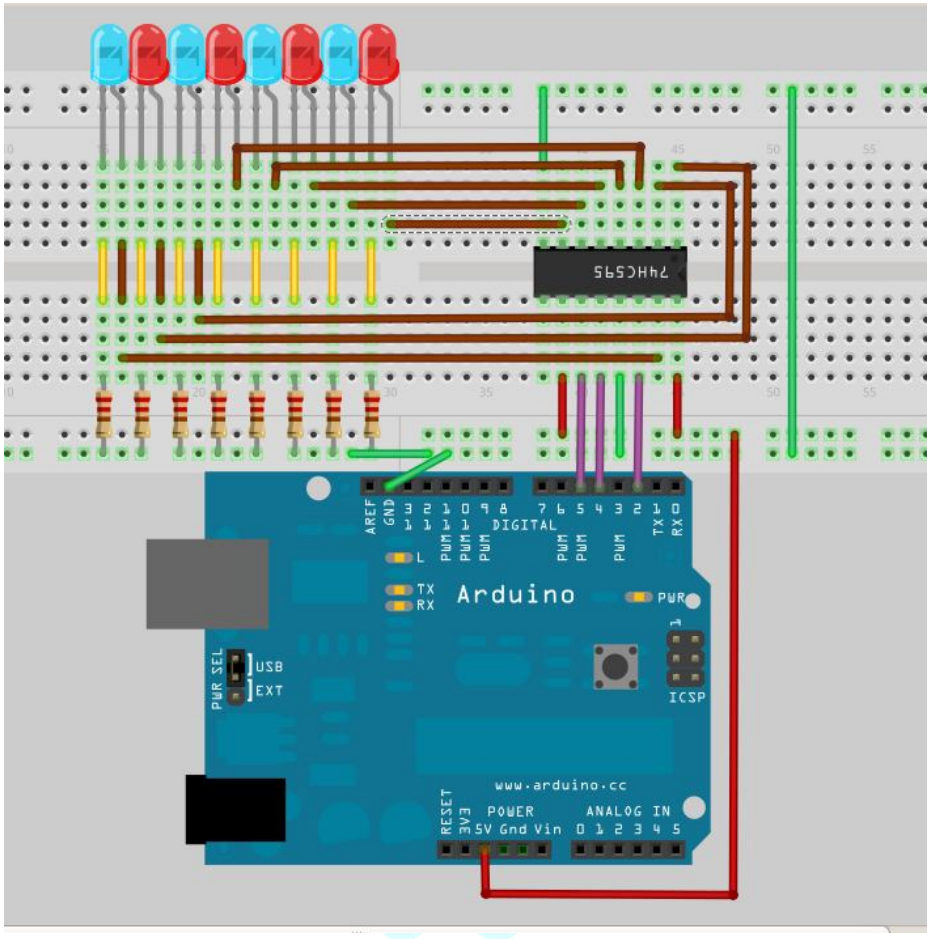


Figure 15.4 Circuit for the LED light pattern sketch.

```
/*
   74HC595 Sketch

   Sends on/off patterns to the LEDs
   via serial to parallel interface
   to the 74HC595 I.C.
*/

const int ON =  HIGH ;
const int OFF = LOW ;

int latchPin = 5;
int clockPin = 4;
int dataPin  = 2;

int delayTime      = 1000; // 1 second delay between LED patterns
int bitPattern     = 0;
int numBitPatterns = 12;   // number of bit patterns in bits[] array

int bits[] = {B00000000,B00000001,B00000010,B00000100,B00001000,B00010000,
B00100000,B01000000,B10000000,B11111111,B01010101, B10101010};


void setup() {
  pinMode(latchPin, OUTPUT);
  pinMode(clockPin, OUTPUT);
  pinMode(dataPin, OUTPUT);
  digitalWrite(clockPin,LOW);
  digitalWrite(latchPin,LOW);
}

void loop() {
  updateLEDs(bitPattern);
  delay(delayTime);
  bitPattern++;
  bitPattern=bitPattern%numBitPatterns;
}
```

Figure 15.5a LED light pattern sketch code - main section.

```
void updateLEDs(int value)
{
  unsigned int bit = bits[value];

  for(int i=0;i<8;i++)
  {

    if((bit & B10000000)==128)
    {
        digitalWrite(dataPin,HIGH);
    }
    else
    {
        digitalWrite(dataPin,LOW);
    }
    digitalWrite(latchPin,HIGH);
    delay(2);
    digitalWrite(latchPin,LOW);
    bit = bit<<1;
  }
  digitalWrite(clockPin,HIGH);
  delay(2);
  digitalWrite(clockPin,LOW);
  digitalWrite(latchPin,LOW);
}
```

Figure 15.5b LED light pattern sketch code – updateLEDs() function.

Chapter 16: 8x8 Dot LED Matrix

The 8x8 dot LED matrix is composed of, as its name implies, 64 individual LEDs. As Figure 16.1 illustrates, all the anodes of the LEDs in a particular column share a common line/pin, and all the cathodes of the LEDs in a particular row share a common line/pin.

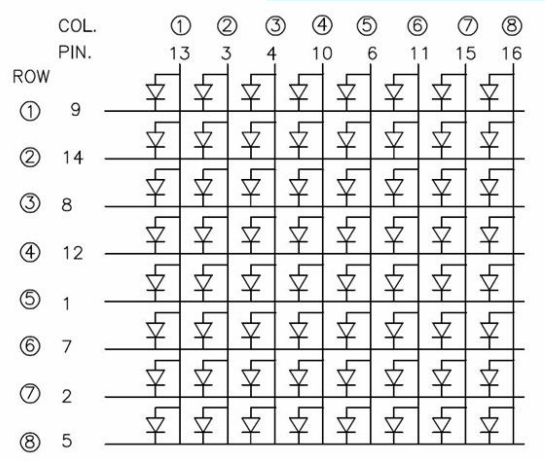


Figure 16.1 8x8 Dot LED matrix display schematic.

Letter display sketch

Sketch components

- 8x8 Dot LED matrix display
- 8 x 220 Ω resistors
- Uno, breadboard, & jumper wires

Connection

The circuit connection for this sketch is shown in Figure 16.2. You will need to be a little creative to make these connections since the 8x8 Dot Matrix does not leave any breadboard holes exposed when plugged into it. If you have two breadboards on hand, then you can connect one side of the Dot Matrix display into one of the breadboards, and the other side of the Dot matrix display into the other breadboard. This would be the optimum configuration for pin access.

Sketch code

The code for this sketch can be uploaded to your Uno from the SainSmart library, under Chapter 16. See Figure 16.3 for example portion of code.

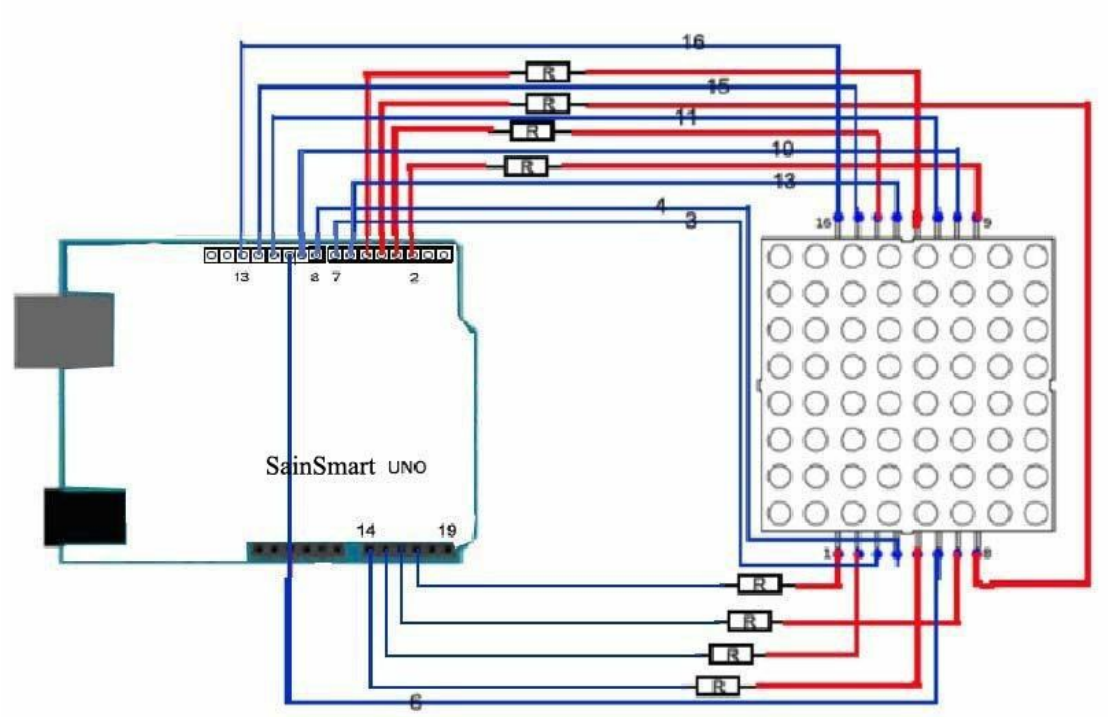


Figure 16.2 Wiring for the letter display sketch.

```
#define data_ascii_A 0x02, 0x0C, 0x18, 0x68, 0x68, 0x18, 0x0C, 0x02 /*"A", 0*/
/**
 **"A"
 #define A  { //
    {0, 0, 0, 0, 0, 0, 1, 0}, //0x02
    {0, 0, 0, 0, 1, 1, 0, 0}, //0x0C
    {0, 0, 0, 1, 1, 0, 0, 0}, //0x18
    {0, 1, 1, 0, 1, 0, 0, 0}, //0x68
    {0, 1, 1, 0, 1, 0, 0, 0}, //0x68
    {0, 0, 0, 1, 1, 0, 0, 0}, //0x18
    {0, 0, 0, 0, 1, 1, 0, 0}, //0x0C
    {0, 0, 0, 0, 0, 0, 1, 0}  //0x02
}
```

Set the value to 1 , then the Led will be turn on !

Figure 16.3 Portion of the sketch code. Demonstrates how the letter A is defined and implemented.

Chapter 17: Infrared Remote Control

Infrared receiver module

Infrared remote controls, such as those that control TVs, DVD players, stereos, cable boxes, etc., send a series of binary pulse codes using infrared light (IR). This is accomplished by the logic inside the remote turning on and off a transmitter IR LED at a set frequency (typically in the tens of thousands of times per second) and with a set coding scheme (protocol). The IR receiver module in the device to be controlled must be tuned to the same carrier frequency as the transmitter and understand the protocol of the data being sent. Additionally, IR transmission/reception only works for line of sight (i.e., the remote has to be pointed at the receiver). The IR transmitting beam will spread, but it is strongest (has more energy) towards the center of the beam, similar to a flashlight. More on the protocol and how this data can be captured by the IR receiver module (Figure 17.1) will be addressed in the example sketch.

Figure 17.1 IR receiver module.

Infrared remote control sketch

Sketch components

- 1 x IR remote control
- 1 x Infrared receiver module
- 1 x LED (any color)
- 1 x 220 ohm resistor

- Uno, breadboard, & jumper wires

Connections (as shown in Figure 17.2)

Connect the receiver module's three pins as follows (check Figure 17.1 for labels):

- VOUT (pin 1) connects to the Uno's digital I/O port/pin 2.
- GND (pin 2) connects to Uno's GND.
- VCC (pin 3) connects to Uno's +5 v source.

Connect Uno's pin 9 to source a series LED and 220 ohm resistor, which is connected to GND.

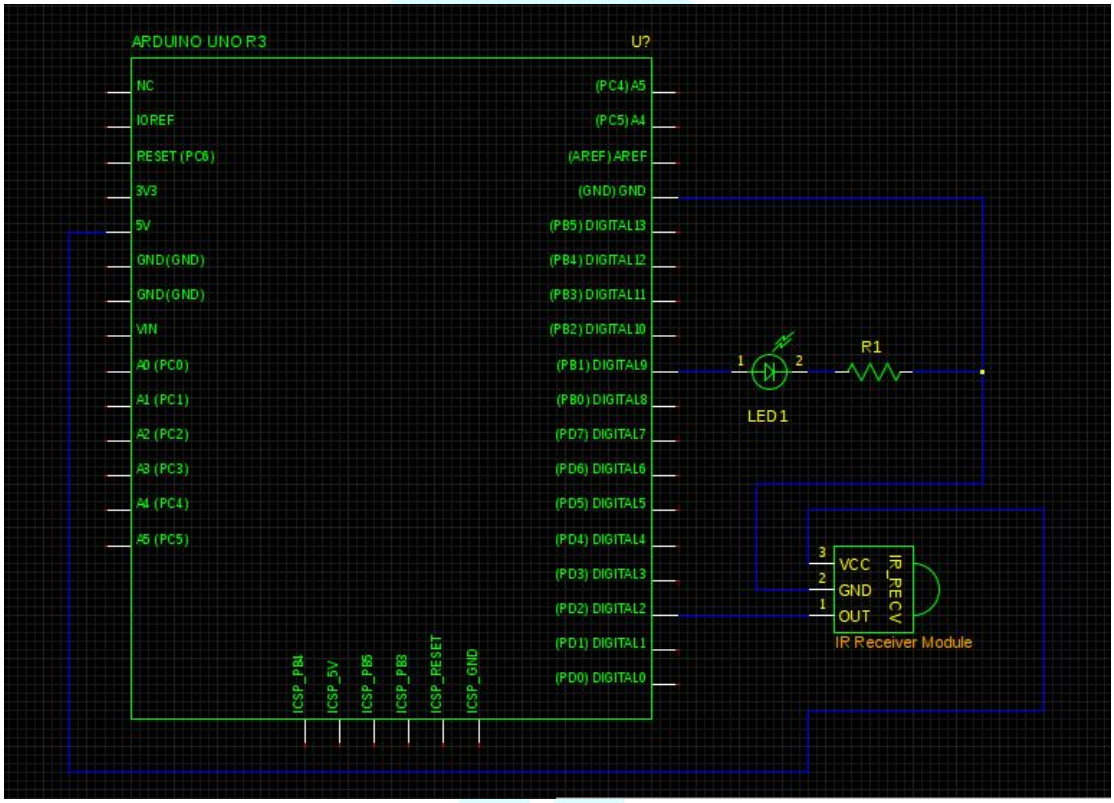


Figure 17.1 IR remote control sketch circuit.

Required files for the sketch

The code for this sketch can be uploaded to your Uno from the SainSmart library. The sketch: *irRemoteDecode.ino* needs an additional, special library named *IRremote* to compile and work properly. The files in this library have functions that take care of the modulation/demodulation of the IR signals and has several protocols it can recognize. This library can be found at the github link under: http://www.righto.com/2009/08/multi-protocol-infrared-remote-library.html. It is also included under the SainSmart library for Chapter 17, along with the *irRemoteDecode* arduino code.

To install this library from the Website, download it, unzip it, and extract the top folder, or if using the *IRremote* folder under SainSmart's chapter 17 directory, copy and paste it, into the arduino IDE's *libraries* folder, located under the folder your arduino program is installed (e.g., Linux: ~/arduino-1.0.5/libraries/ or Windows: >

Computer > Local Disk(C:) > Program Files (x86) > Arduino > libraries). If you have the Arduino IDE already up before you install this library, you will need to close it and re-launch it.

Sketch description

Once you have upload the *irRemoteDecode* sketch to the Uno, open the Serial Monitor window by clicking on the button in the upper right hand corner of the Arduino IDE using your mouse. Make sure the baud rate in the lower right hand corner of this window is set to 9600. The sketch will then prompt you in the Serial Monitor window to press buttons 0 thru 4 of the small NEC remote, while pointing it at the IR receiver module. Once the sketch has "learned" the codes for these buttons, you can change the brightness of the LED with buttons 1 thru 4, and turn off the LED with button 0.

NEC protocol described

Feature:

（1）8-bit address spaces, 8-bit command spaces.
（2）Address bits and command bits are transmitted twice for reliability.
（3）Pulse distance modulation.
（4）Carrier frequency 38 kHz.
（5）Every bit's time is 1.125ms or 2.25ms.

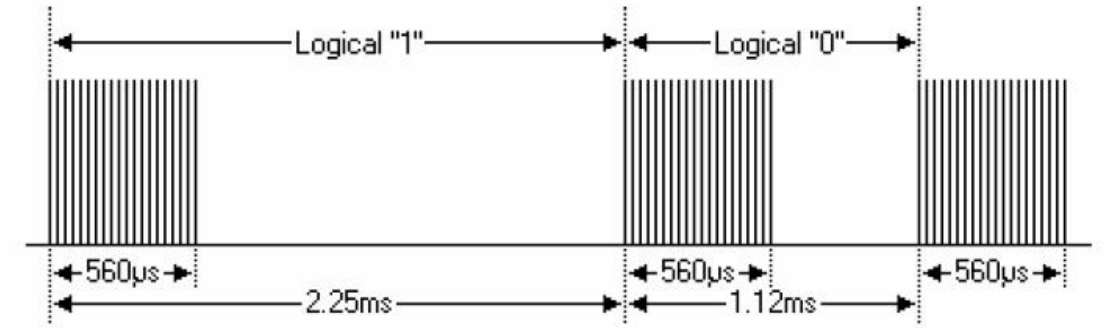


Figure 17.2 A logic "1" and logic "0" for the NEC protocol.

Protocol:

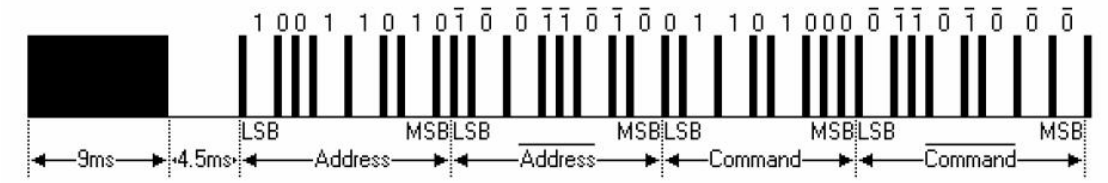


Figure 17.3 Example NEC protocol message.

As can be seen in Figure 17.2, the logic "1" takes twice as long to transmit as a logic "0". The beginning of each logic bit is started with a "mark". The mark produced by turning on and off the IR LED at the carrier frequency of 38 kHz.

Figure 17.3 shows the typical NEC protocol pulse sequence. As you can see from this figure, the LSB (least significant bit) is sent first. In this example, the Address 0x59

and the Command 0x16 are being transmitted. A message is start by a 9ms AGC (automatic gain control) burst or “mark”, followed by a 4.5ms “space”, then by the address and command. The second time all bits are inverted and can be used for verification. Thus, the total transmission time is constant, do to the duplication of every bit and its inverted length.

Chapter 18: Liquid Crystal Display

What is the 1602 LCD?

The 1602 LCD (liquid crystal display), as show in Figure 18.1, is compatible with the Hitachi HD44780 controller. This is important because the LiquidCrystal library that comes with the Arduino IDE emulates this type of controller, and hence is able to drive the 1602 LCD module via its library functions.

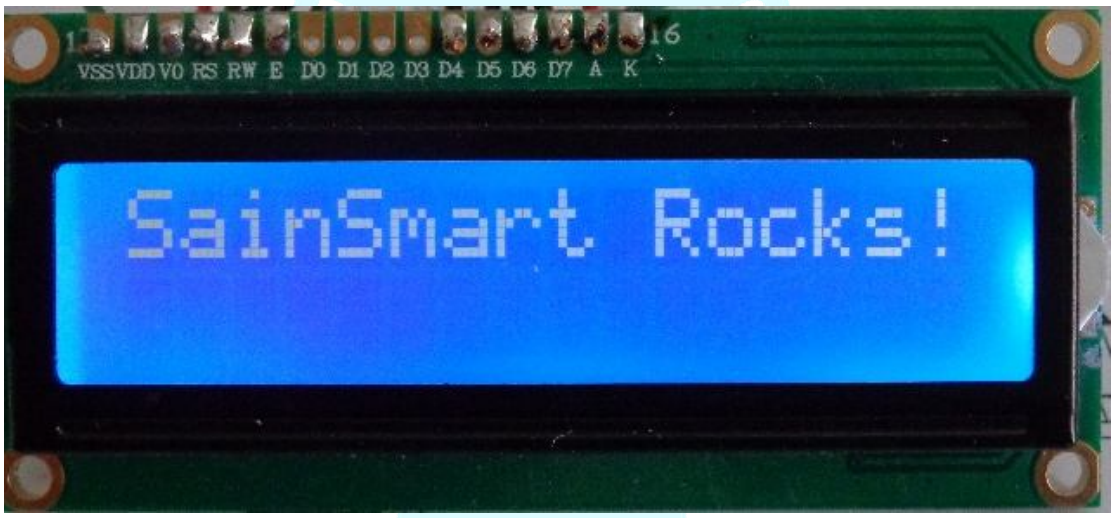


Figure 18.1 The 1602 LCD module

Some of the 1602 LCD functional characteristics are as follows:

Display capacity: 16x2 characters;
Chip operating voltage: 4.5V~5.5V;
Operating current: 2.0mA(5.0V);
Best operating voltage: 5.0V;
Character size: 2.95x4.35(WxH) mm.

The functionality of each pin is described in Table 18.1. As you can see, there are two sets of power that gets supplied to the LCD. Pins 1 & 2 supply power to the basic functionality of the module, while pins 15 & 16 are for the LCD's backlighting. Additionally, using the center tap of a potentiometer to adjust the voltage level to the LCD's V0 pin, allows for the contrast of the LCD display to be adjusted.

To be able to interface to the LCD, you will need to solder a 16 pin header, if you have one, or 12 wires (first 6 pins and last 6 pins) to this module. If you are using wires to solder to the LCD, make sure that they are of thick enough gauge to plug into a breadboard or the Uno header.

Table 18.1 Pin Assignments and Description

No.	Symbol	Level	Function	
1	VSS	--	0v	Power Supply
2	VDD	--	+5v	
3	V0	--	for LCD	
4	RS	H/L	Register Select: H:Data Input, L:Instruction Input	
5	R/W	H/L	H = Read, L = Write	
6	E	H, H-L	Enable Signal	
7	D0	H/L	Data bus used in 8 bit transfer	
8	D1	H/L		
9	D2	H/L		
10	D3	H/L		
11	D4	H/L	Data bus for both 4 and 8 bit transfer	
12	D5	H/L		
13	D6	H/L		
14	D7	H/L		
15	BL-A	--	Backlight +5v	
16	BL-K	--	Backlight 0v	

Table 18.2 Basic Operation

Read status	Input	RS=L, R/W=H, E=H	Output	D0~D7=status word
Write command	Input	RS=L, R/W=L, D0~D7=command code, E= high pulse	Output	none
Read data	Input	RS=H, R/W=H, E=H	Output	D0~D7=data
Write data	Input	RS=H, R/W=L, D0~D7=data, E= high pulse	Output	none

Basic operation description (from: http://arduino.cc/en/Tutorial/LiquidCrystal)

The LCDs have a parallel interface, meaning that the microcontroller has to manipulate several interface pins at once to control the display. The interface consists of the following pins:

A **register select (RS) pin** that controls where in the LCD's memory you're writing data to. You can select either the data register, which holds what goes on the screen, or an instruction register, which is where the LCD's controller looks for instructions on what to do next.

A **Read/Write (R/W) pin** that selects reading mode or writing mode

An **Enable pin** that enables writing to the registers

8 **data pins (D0 -D7)**. The states of these pins (high or low) are the bits that you're writing to a register when you write, or the values you're reading when you read.

The process of controlling the display involves putting the data that form the image of what you want to display into the data registers, then putting instructions in the instruction register. The LiquidCrystal Library simplifies this for you so you don't need to know the low-level instructions.

The Hitachi-compatible LCDs can be controlled in two modes: 4-bit or 8-bit. The 4-bit mode requires seven I/O pins from the Arduino, while the 8-bit mode requires 11 pins. For displaying text on the screen, you can do most everything in 4-bit mode, so the circuit in Figure 18.2 shows how to control a 2x16 LCD in 4-bit mode.

Circuit for sketches (Figures 18.2 & 18.3)

To wire your LCD screen to your Arduino, connect the following pins:

LCD RS pin to digital pin 12
LCD Enable pin to digital pin 11
LCD D4 pin to digital pin 5
LCD D5 pin to digital pin 4
LCD D6 pin to digital pin 3
LCD D7 pin to digital pin 2

Additionally, wire a 10K pot to +5V and GND, with it's wiper (output) to LCD screens V0 pin (pin3), and the backlight pins: A to+5v, and K to ground (not shown connected in these figures).

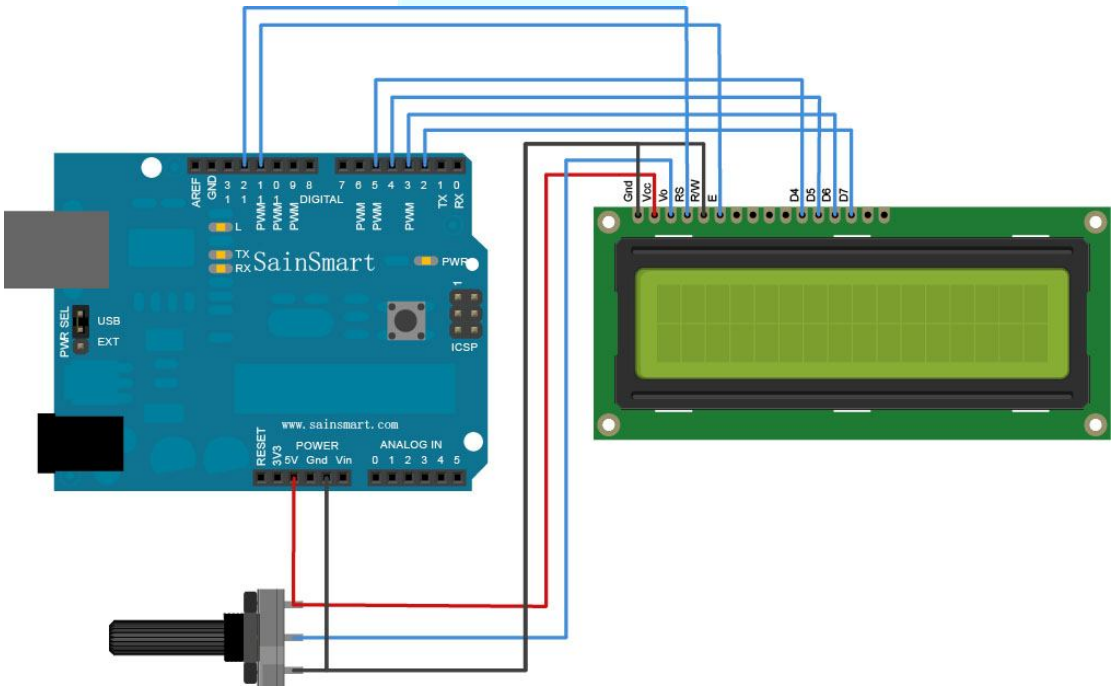


Figure 18.2 Drawing of the 1602 LCD and Uno connection for the LiquidCrystal sketches.

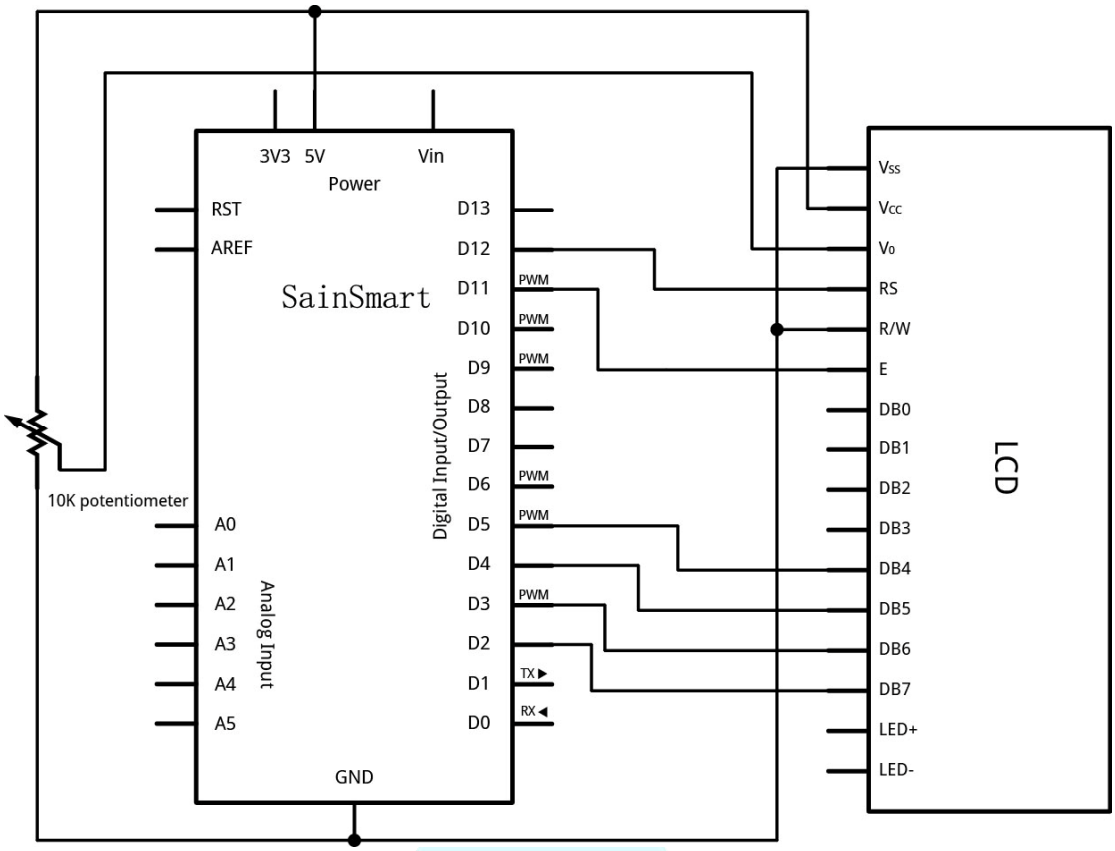


Figure 18.3 Schematic of 1602 LCD and Uno for the LiquidCrystal sketches.

LiquidCrystal sketches

The Arduino program software you downloaded comes with sample sketches to run the circuit shown in Figures 18.2 and 18.3. They all use the same pinouts. The sketches can be found from the IDE, under **File->Examples->LiquidCrystal**. It is suggested that you start with the simple HelloWorld sketch to ensure your connections and pot setting are correct. Go ahead and modify this sketch to display something else besides "hello, world!". Just remember that if you wish to save your changes, you will need to do a "save as" and place the modified sketch in your Sketchbook. There are some cool sketches that show how you can move your message back and forth on the screen, and more!

Chapter 19: Opto-Isolated, 2 Channel Relay Module

What is a relay?

A relay is an electric switch. It is used to isolate a low power controlling side from a higher power connecting side. This is normally accomplished by use of an electromagnet, which controls the connectivity of a center pole contact with two separate contacts. When the electromagnet is off, connection between the center pole and one of the contacts (normal closed contact – NC) is made and sustained. When the electromagnet is on, it attracts the center pole away from the default connector, therefore disconnecting the two contacts, and forms a new connection with the other connector or normal open contact (NO).

The project in this chapter will use a 5V, 2-Channel Relay interface board, which is pictured in Figure 19.1. The relay board schematic, which shows one channel (one relay), is illustrated in Figure 19.2.

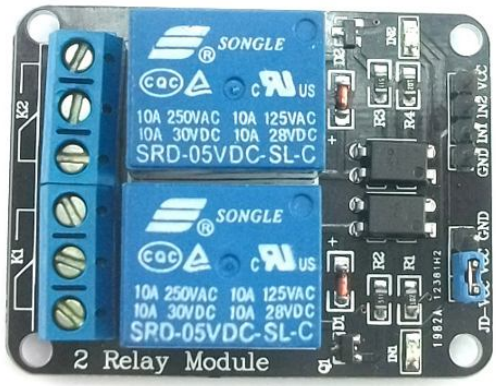


Figure 19.1 Opto-Isolated, 2 Channel Relay Module.

Product features:

- 5V, 2-Channel Relay interface board, where each channel requires 15-20mA driver current.
- 2 levels of input isolation: opto-isolator chip, and the relay.
- Equipped with high-current relay, AC250V 10A ; DC30V 10A
- Standard interface that can be controlled directly by microcontroller (Arduino , 8051, AVR, PIC, DSP, ARM, ARM, MSP430, TTL logic)
- Indication LED's for Relay output status

How to use this module.

Apply Power

Find the 4-pin male connector on the relay module (see top right hand side of module in Figure 19.1). Connect the GND pin of this connector to one of the Uno's ground pins. Then connect the VCC pin of this connector to one of the Uno's V (of the S, V, G pins) or +5v pin. Set the Uno to operate at 5v, since this is the voltage level the relay works at. Keep the jumper connecting JD-VCC pin to VCC pin on the relay board's jumper. This allows for the Uno to supply both of the opto-isolator chips and the two relay switches with power, see Figure 19.2 for details. If you are going to connect two or more of these modules to a single Uno, then you would want to remove this jumper and supply the JD-VCC and common ground with a separate +5v power supply. This would be needed because each relay draws 80 mA of current when switched on, besides another 20 mA for each input to turn on the internal LED of the opto-isolator chip, so several of these would put a drain on the USB connection, if you are using a computer for supplying power to the Uno.

Relay Off Mode

The opto-isolated, 2 channel relay module has 2 levels of input isolation. The first level or stage is through an opto-isolator, NPN transistor semiconductor (U1 in Figure 19.2). When IN1 is connected to Vcc, the LED inside U1 is off, so the NPN transistor coupled with that LED is also off. Therefore, the transistor in U1 is acting like an open circuit and will not allow current to pass through it. This, in turn, causes Q1 to be off and act like an open circuit as well. The second level of isolation is caused by the relay itself, which isolates the control of the relay from the circuit resulting from the connection make by the relay. Figure 19.3 illustrates the internal state of the relay off mode.

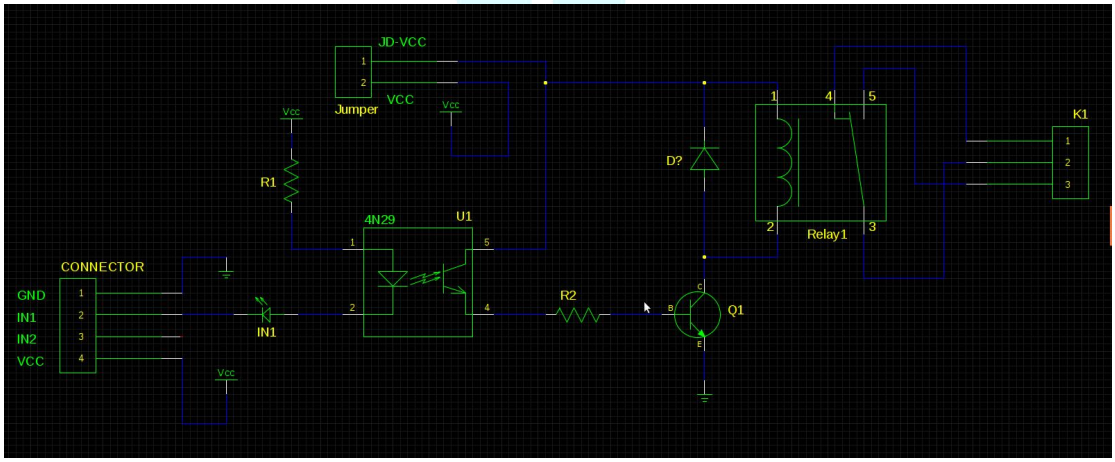


Figure 19.2 Schematic of 1-channel of the 2-channel relay board.

Relay On Mode

When the IN1 pin of the 4-pin connector is grounded or brought to ground, current will flow through R1, the internal LED in U1, and the IN1 LED (which lights the red

IN1 LED). With the U1 internal LED on, this turns on the NPN transistor in U1, which turns on Q1. This causes current to flow through the electromagnet of the relay, which disconnects the NC contact and causes a connection with the NO contact.

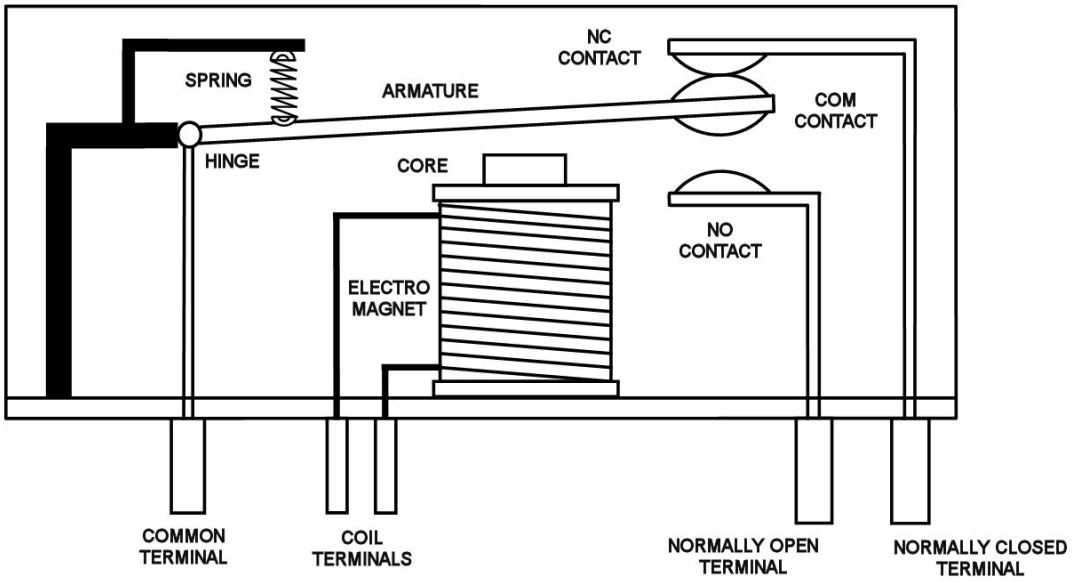


Figure 19.3 The internals of a two contact relay.
(Source: http://www.glolab.com)

Multimeter test of relays

Since relays are typically used for high power applications (e.g., making connections with high voltage and/or current sources), and is not safe for the average hobbyist, thus we describe a simple test of the relays. This test can be performed with or without the use of the Uno's I/O pins. All that is needed is power to the relay board, as described in the previous "**Apply Power**" section. By connecting the IN1 and IN2 pins to either ground (relay on: NO connects to common terminal) or Vcc (relay off: NC connects to common terminal), verify the connectivity between the common terminal with the NO and NC terminals with a multimeter or the two leads from a simple break in an LED circuit.

DISCLAIMER

SainSmart is not responsible for any injury and/or death that may occur to anyone connecting this part to high power circuits. Such use needs to be done by a licensed electrician and/or someone who has experience with high power circuits and are following standard safety procedures.

Chapter 20: Distance Sensor

The ultrasonic ranging module HC-SR04 (Figure 20.1), also referred to as the "Ping" sensor, can measure distances in front of it at a range of 2cm to 400cm, with an accuracy of up to 3mm. The module includes an ultrasonic transmitter, receiver, and control circuit. The HC-SR04 is initiated by being sent a pulse of at least 10µs in duration to its *Trig* pin (assuming that Vcc and Gnd are connected to power). The module will then transmit a train of eight 40 kHz ultrasonic sound waves, while waiting for a return signal on its receiver port. The HC-SR04 will then send a pulse back on its *Echo* pin, whose duration is proportional to the distance detected. A summary of these steps are as follows:

- Using I/O trigger for at least 10µs high level signal,
- The Module automatically sends eight 40 kHz and detect whether there is a pulse signal back.
- If there is a return signal detected, a high level pulse is returned on the echo I/O representing the full flight time of the sound wave.

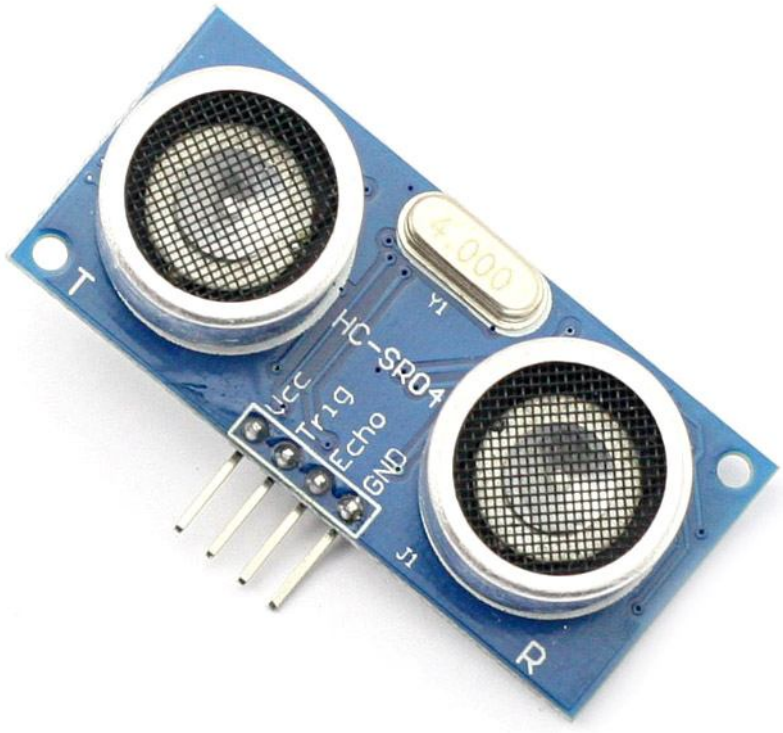


Figure 20.1 Ultrasonic ranging module HC-SR04

The distance of the object detected can be calculated from the return pulse duration returned on the Echo pin. Since the duration represents the time it took from the start of the pulse burst sent to detecting its return at the receiver, the time it took for the

sound wave to travel to the object is ½ of this time. Thus, the formula for converting the total sound wave flight time, given in micro-seconds (µs or 10^{-6} seconds), into the distance to the object detected is as follows:

Object's distance (cm) = (Echo's high level time (µs)/29 (µs/cm))/2 (20.1)

The 29 cm/µs in equation 20.1 represents the velocity of sound waves in air (approximately 340 meters/sec). Dividing the result by 2 gives the one way distance instead of the round trip distance.

You may be wondering how the Uno will know how long the pulse sent back from the HC-SR04 is. Well, someone has written a library of functions that makes this task very simple. We will cover how to install this library, what functions are at your disposal, and how to implement the distance calculation in the Uno in the following sketch.

Distance sketch

Sketch components

- 1 x HC-SR04 ultrasonic ranging module
- Uno, breadboard, & jumper wires

Connections

The connections for this sketch is shown in Figure 20.2 and listed in Table 20.1.

Table 20.1 Distance sketch connection between HC-SR04 and Uno.

HC-SR04 Pins	I/O	Uno's Pins
Vcc	Input	+5v
Trigger	Input	2
Echo	Output	3
Gnd	Input	Ground

Required files for the sketch

The code for this sketch can be uploaded to your Uno from the SainSmart library. The sketch: *pingDistance.ino* needs an additional, special library named *NewPing* to compile and work properly. The files in this library have the following methods (functions) which simplify the interface to the HC-SR04 ultrasonic ranging module (source: http://playground.arduino.cc/Code/NewPing):

Methods

- *sonar.ping*(); - Send a ping, returns the echo time in microseconds or 0 (zero) if no ping echo within set distance limit
- *sonar.ping_in*(); - Send a ping, returns the distance in inches or 0 (zero) if no ping echo within set distance limit

- *sonar.ping_cm*(); - Send a ping, returns the distance in centimeters or 0 (zero) if no ping echo within set distance limit
- *sonar.ping_median*(iterations); - Do multiple pings (default=5), discard out of range pings and return median in microseconds
- *sonar.convert_in*(echoTime); - Converts microseconds to distance in inches
- *sonar.convert_cm*(echoTime); - Converts microseconds to distance in centimeters
- *sonar.ping_timer*(function); - Send a ping and call function to test if ping is complete.
- *sonar.check_timer*(); - Check if ping has returned within the set distance limit.
- *timer_us*(frequency, function); - Call function every frequency microseconds.
- *timer_ms*(frequency, function); - Call function every frequency milliseconds.
- *timer_stop*(); - Stop the timer.

This library can be found under the "*Download NewPing Library*" link on the following Arduino Website: http://playground.arduino.cc/Code/NewPing. It is also included under the SainSmart library for Chapter 20, along with the *pingDistance* arduino code.

To install this library from the Website, download it, unzip it, and extract the top folder, or if using the *NewPing* folder under SainSmart's chapter 20 directory, copy and paste it, into the arduino IDE's *libraries* folder, located under the folder your arduino program is installed (e.g., Linux: ~/arduino-1.0.5/libraries/ or Windows: > Computer > Local Disk(C:) > Program Files (x86) > Arduino > libraries). If you have the Arduino IDE already up before you install this library, you will need to close it and re-launch it.

Sketch description

Once you have upload the *pingDistance* sketch to the Uno, open the Serial Monitor window by clicking on the button in the upper right hand corner of the Arduino IDE using your mouse. Make sure the baud rate in the lower right hand corner of this window is set to 115200. A listing of the code is shown in Figure 20.3 and example data being sent to the Serial Monitor window is shown in Figure 20.4.

In the sketch, the following line uses a call to the NewPing class constructor, which creates an object of this class:

```
NewPing    sonar(TRIGGER_PIN, ECHO_PIN, MAX_DISTANCE);
```

where, parameters: TRIGGER_PIN, ECHO_PIN, and MAX_DISTANCE were defined at the top of the sketch.

Thus, the sonar object knows which pins to send a trigger pulse to, and receive the echo pulse from. Also, you can define a max distance, where a call to one of the ping methods will return a 0, if the distance is equal to or greater than this max amount.

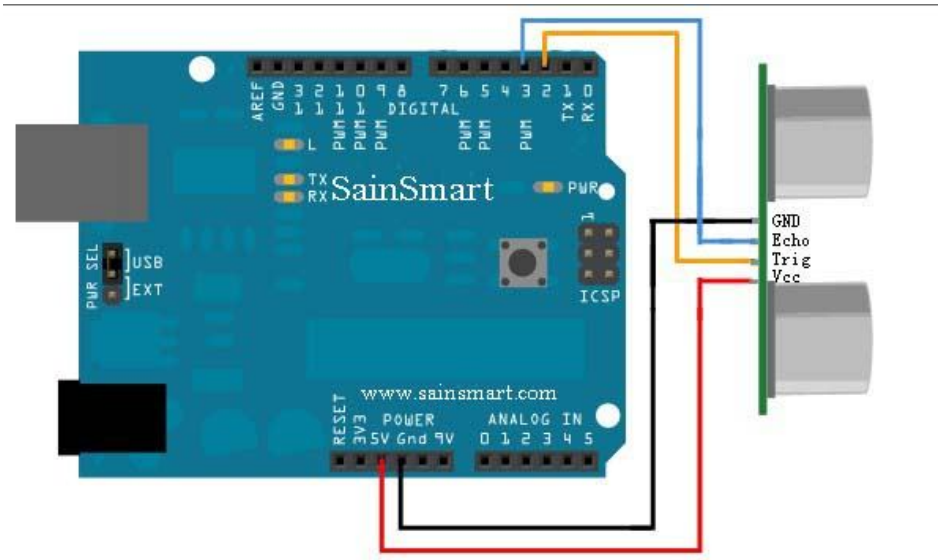


Figure 20.2 Distance sketch connections.

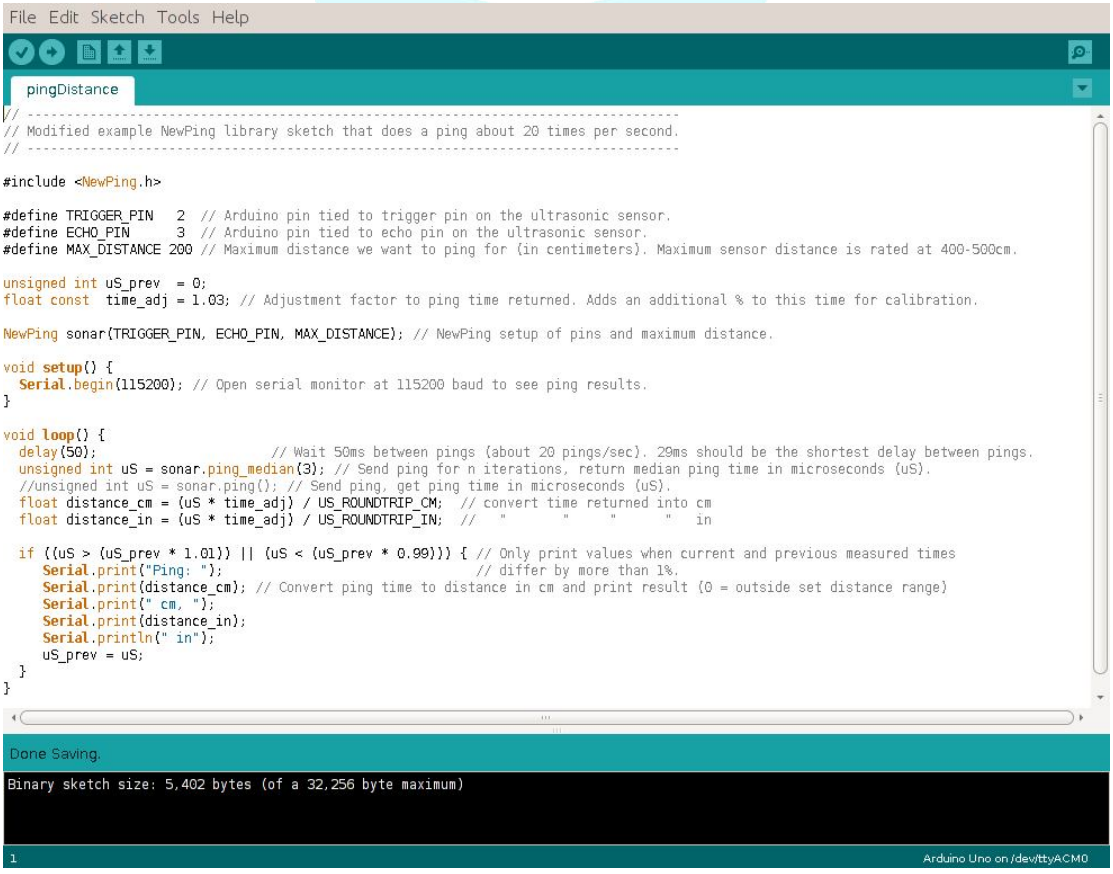


Figure 20.3 Distance sketch.

In the loop() function, we call the sonar object's ping_median(iterations) function. This method, as described in the above "Methods" section, will automatically call the ping() function/method for the number of iterations defined. The method then finds the median of these multiple ping calls and will then return that value in

micro-seconds. This is a form of digital filtering, as it is taking several samples and selecting the median of these values to add accuracy to the measurement.

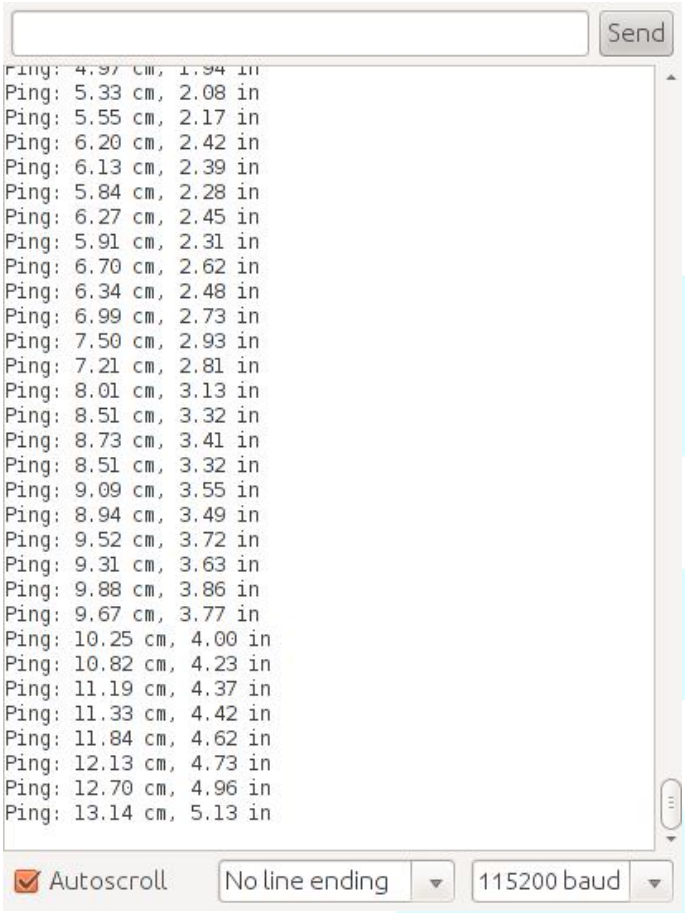


Figure 20.4 Distance sketch data in the Serial Monitor window.

Additionally, a *time_adj* variable has been added to the distance calculation. This allows for you to calibrate your ping sensor for better accuracy. I like putting a ruler stemming out from the bottom of the module and stand a flat piece of cardboard at the various inch markings to see how accurate the ping sensor's measurements are. In the sketch, 3% of the returned sound wave travel time is added on, before being divided by US_ROUNDTRIP_CM/IN, because my distance was coming up short. This may be different for your sensor, so adjust accordingly. It isn't needed if you are not concerned with pinpoint accuracy, such as collision avoidance applications.

You may also notice that it appears that we have forgotten to divide the distance by 2 to get the distance to the object and not the round trip distance. However, this division is figured into the US_ROUNDTRIP_CM/IN constants.

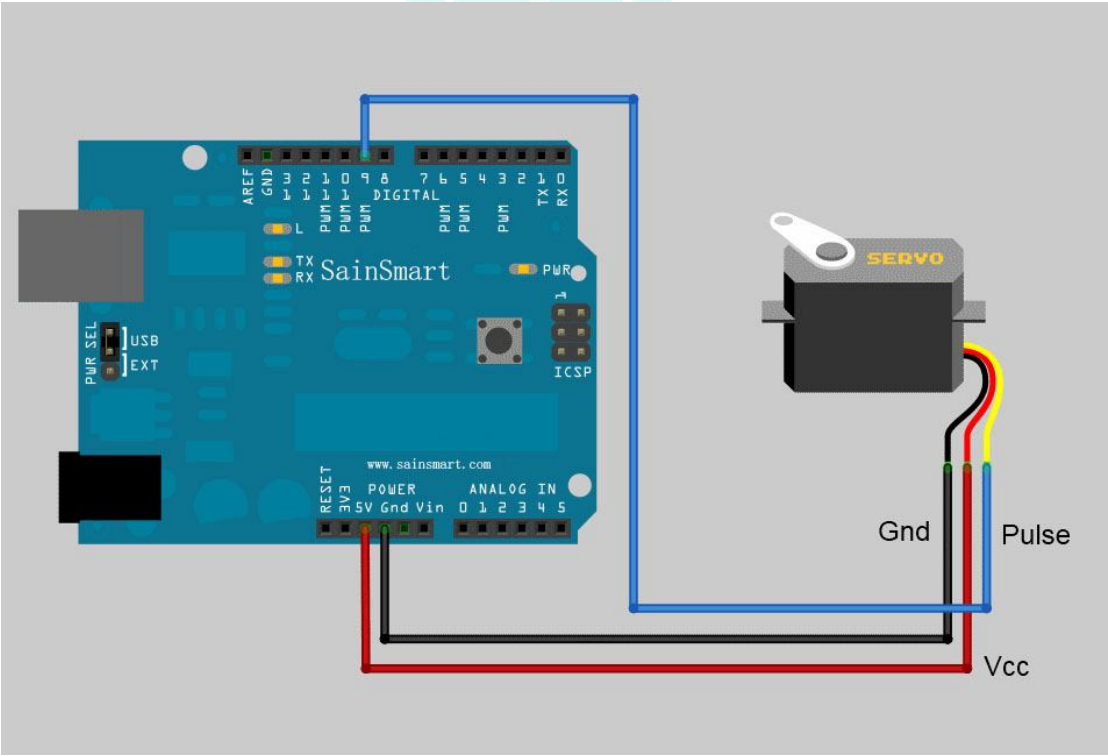
An if statement was added to cut down on the scrolling of the values on the Serial Monitor. The if statement checks to see if the newly measured roundtrip time is greater than 1% of the previously measured time, before it will print out the results. Feel free to make up your own print criterion!

Chapter 21: Servo Motor

Controlling a servo motor with an Arduino or other type of microcontroller is probably the easiest way to get started in robotics, motion art, or any other reason you may have to make your electronic project interact with the real world. Servos are very simple to interact with and in this post I'll show you how to connect one to an Arduino.

Servo motors are a specific type of motor, often used in hobby RC cars and planes, that rotate to a specific angle when a corresponding signal is applied to the pulse pin. Servo motors are very easy to program and very strong for their size. This makes them useful for a wide array of applications. The internal components of a servo motor consist of a regular DC motor, which does the actual work, a system of gears to increase the torque to the output shaft, and a circuit board and sensors to control the movement of the motor.

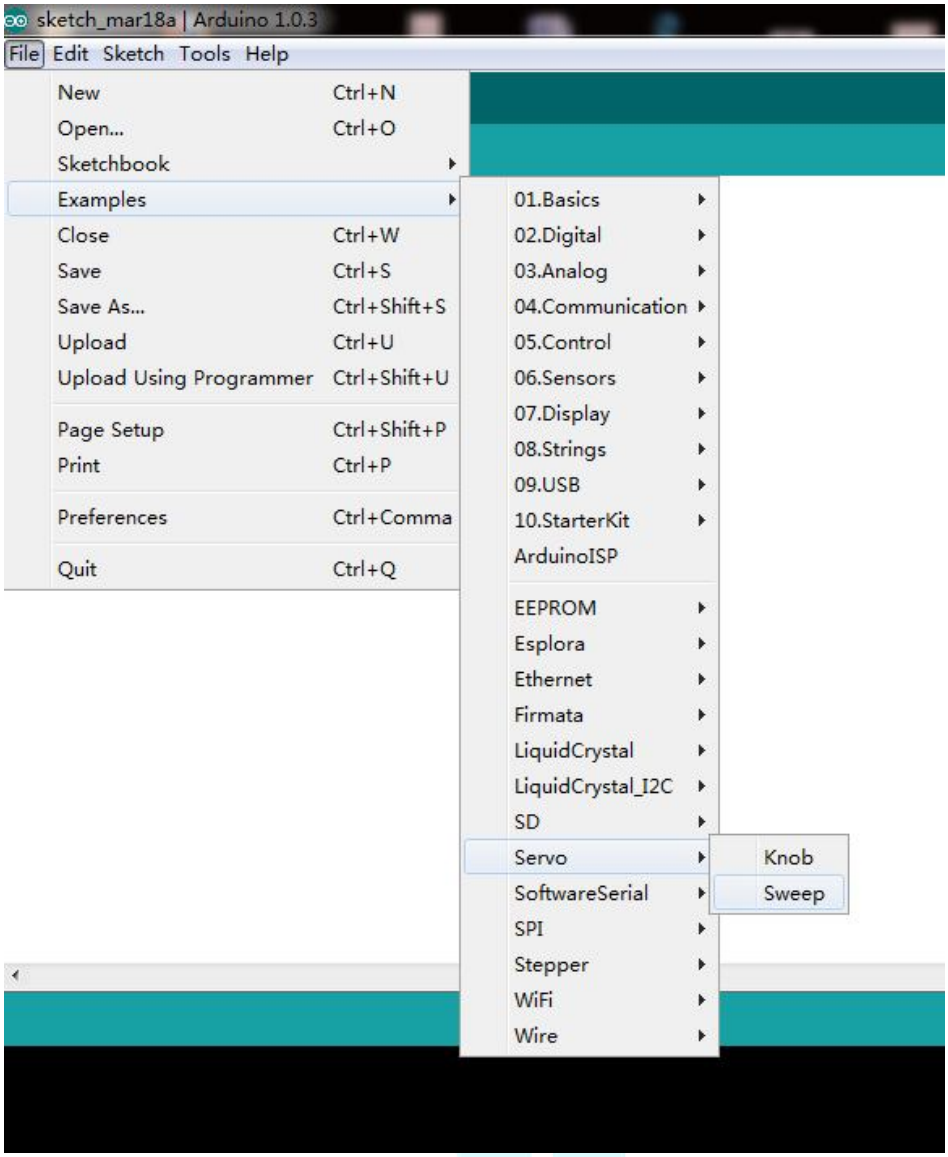
Wiring:
To get started controlling a servo with your Arduino, you only need to **connect three pins**. There are **two pins for power and ground**. For a small servo or just for testing, you can connect these directly to the Arduino. If you are controlling a large servo motor, you might want to use an external power source. Just remember to connect the ground from the external source to the ground of the Arduino.



The third pin is the **pulse**, or **signal** pin. This accepts the signal from your controller that tells it what angle to turn to. The control signal is fairly simple compared to that of a stepper motor. It is just a pulse of varying lengths. The length of the pulse corresponds to the angle the motor turns to. Typically a pulse of **1.25 milliseconds causes the motor to rotate to 0 degrees** and a pulse of **1.75 milliseconds turns it 180 degrees**. Any length of pulse in between will rotate the servo shaft to its corresponding angle. **Some servos will turn more or less than 180 degrees**, so you may need to experiment.

Programming:

The Arduino software comes with a sample servo sketch and servo library that will get you up and running quickly. Simply load it from the menu as shown below. Their example uses pin 9 for the pulse wire, so to keep it simple, that's what I used. You could use any of the data pins and, if you add more than one servo, you will need to. The Sweep sample simply rotates the servo back and forth from 0 degrees to 180. There is another sample sketch that uses a potentiometer as an input to control the angle of the motor, but I'll get in to that later.



The code is pretty basic and well documented. It first loads the library needed and sets up which pin to use as the output.

This line tells it to move from 0 degrees to 180 degrees one degree at a time:

for(pos = 0; pos < 180; pos += 1)

And this line tells it to move back to 0 degrees one degree at a time.

for(pos = 180; pos>=1; pos-=1)

Chapter 22: XBee shield

What is the XBee shield?

The Arduino XBee shield (an Expansion Board without XBee module) is a compliant solution designed to meet low-cost, low-power wireless sensor networks with special needs. The module is easy to use, low power consumption, and the provision of critical data between devices reliable transmission. As the innovative design, XBee-PRO can be in the range 2-3 times beyond the standard ZigBee modules. XBee-PRO modules work in the ISM 2.4 GHz frequency band. The MaxStream's XBee (1 mW) Zigbee module is pin-compatible.

The Xbee module is widely used in the United States, Canada, Australia, Israel and Europe. The establishment of RF communication does not require any configuration and the module's default configuration supports a wide range of data system applications. You can also use a simple AT command to advanced configuration. An OEM developer is now XBee code development package. It is self-developed in collaboration with the MaxStream ZigBee/802.15.4 RF module code.

Example

You should be able to get two SainSmart boards with XBee shields talking to each other without any configuration, using just the standard SainSmart serial commands.

To upload a sketch to an SainSmart board with a XBee shield, you'll need to put both jumpers on the shield to the "USB" setting (i.e. place them on the two pins closest to the edge of the board) or remove them completely (but be sure not to lose them!). Then, you can upload a sketch normally from the SainSmart environment. In this case, upload the Communication | Physical Pixel sketch to one of the boards. This sketch instructs the board to turn on the LED attached to pin 13 whenever it receives an 'H' over its serial connection, and turn the LED off when it gets an 'L'. You can test it by connecting to the board with the SainSmart serial monitor (be sure it's set at 9600 baud), typing an H, and pressing enter (or clicking send). The LED should turn on. Send an L and the LED should turn off. If nothing happens, you may have an SainSmart board that doesn't have a built-in LED on pin 13.

Once you've uploaded the Physical Pixel sketch and made sure that it's working, unplug the first SainSmart board from the computer. Switch the jumpers to the XBee setting (i.e. place each on the center pin and the pin farthest from the edge of the board). Now, you need to upload a sketch to the other board. Make sure its jumpers are in the USB setting. Then upload the following sketch to the board:

```
void setup()
{
Serial.begin(9600);
}

void loop()
{
Serial.print('H');
delay(1000);
Serial.print('L');
delay(1000);
}
```

When it's finished uploading, you can check that it's working with the SainSmart serial monitor. You should see H's and L's arriving one a second. Turn off the serial monitor and unplug the board. Switch the jumpers to the XBee setting. Now connect both boards to the computer. After a few seconds, you should see the LED on the first board turn on and off, once a second. (This is the LED on the SainSmart board itself, not the one on the XBee shield, which conveys information about the state of the XBee module.) If so, congratulations, your SainSmart boards are communicating wirelessly. This may not seem that exciting when both boards are connected to the same computer, but if you connect them to different computers (or power them with an external power supply - being sure to switch the power jumper on the SainSmart board), they should still be able to communicate.

Addressing

There are multiple parameters that need to be configured correctly for two modules to talk to each other (although with the default settings, all modules should be able to talk to each other). They need to be on the same network, as set by the ID parameter (see "Configuration" below for more details on the parameters). The modules need to be on the same channel, as set by the CH parameter. Finally, a module's destination address (DH and DL parameters) determine which modules on its network and channel will receive the data it transmits. This can happen in a few ways:

- If a module's DH is 0 and its DL is less than 0xFFFF (i.e. 16 bits), data transmitted by that module will be received by any module whose 16-bit address MY parameter equals DL.
- If DH is 0 and DL equals 0xFFFF, the module's transmissions will be received by all modules.
- If DH is non-zero or DL is greater than 0xFFFF, the transmission will only be received by the module whose serial number equals the transmitting module's destination address (i.e. whose SH equals the transmitting module's DH and whose SL equals its DL).

Again, this address matching will only happen between modules on the same network and channel. If two modules are on different networks or channels, they can't communicate regardless of their addresses.

Configuring the XBee module

You can configure the XBee module from code running on the SainSmart board or from software on the computer. To configure it from the SainSmart board, you'll need to have the jumpers in the Xbee position. To configure it from the computer, you'll need to have the jumpers in the USB configuration and have removed the microncontroller from your SainSmart board.

To get the module into configuration mode, you need to send it three plus signs: +++ and there needs to be at least one second before and after during which you send no other character to the module. Note that this includes newlines or carriage return characters. Thus, if you're trying to configure the module from the computer, you need to make sure your terminal software is configured to send characters as you type them, without waiting for you to press enter. Otherwise, it will send the plus signs immediately followed by a newline (i.e. you won't get the needed one second delay after the +++). If you successfully enter configuration mode, the module will send back the two characters 'OK', followed by a carriage return.

Send Command	Expected Response
+++	OK<CR>

Once in configuration mode, you can send AT commands to the module. Command strings have the form ATxx (where xx is the name of a setting). To read the current value of the setting, send the command string followed by a carriage return. To write a

new value to the setting, send the command string, immediately followed by the new setting (with no spaces or newlines in-between), followed by a carriage return. For example, to read the network ID of the module (which determines which other XBee modules it will communicate with), use the 'ATID command:

Send Command	**Expected Response**
ATID<enter>	3332<CR>

To change the network ID of the module:

Send Command	**Expected Response**
ATID3331<enter>	OK<CR>

Now, check that the setting has taken effect:

Send Command	**Expected Response**
ATID<enter>	3331<CR>

Unless you tell the module to write the changes to non-volatile (long-term) memory, they will only be in effect until the module loses power. To save the changes permanently (until you explicitly modify them again), use the **ATWR** command:

Send Command	**Expected Response**
ATWR<enter>	OK<CR>

To reset the module to the factory settings, use the **ATRE** command:

Send Command	**Expected Response**
ATRE<enter>	OK<CR>

Note that like the other commands, the reset will not be permanent unless you follow it with the **ATWR** command.

Here are some of the more useful parameters for configuring your XBee module.

Command	Description	Valid Values	Default Value
ID	The network ID of the XBee module.	0 - 0xFFFF	3332
CH	The channel of the XBee module.	0x0B - 0x1A	0X0C
SH and SL	The serial number of the XBee module (SH gives the high 32 bits, SL the low 32 bits). Read-only.	0 – 0xFFFFFFFF (for both SH and SL)	different for each module
MY	The 16-bit address of the module.	0 - 0xFFFF	0
DH and DL	The destination address for wireless communication (DH is the high 32 bits, DL the low 32).	0 – 0xFFFFFFFF (for both DH and DL)	0 (for both DH and DL)
BD	The baud rate used for serial	0 (1200 bps) 1 (2400 bps)	3 (9600 baud)

	communication with the Arduino board or computer.	2 (4800 bps) 3 (9600 bps) 4 (19200 bps) 5 (38400 bps) 6 (57600 bps) 7 (115200 bps)	

Note: although the valid and default values in the table above are written with a prefix of "0x" (to indicate that they are hexadecimal numbers), the module will not include the "0x" when reporting the value of a parameter, and you should omit it when setting values.

Here are a couple more useful commands for configuring the Xbee module (you'll need to prepend AT to these too).

Command	Description
RE	Restore factory default settings (note that like parameter changes, this is not permanent unless followed by the WR command).
WR	Write newly configured parameter values to non-volatile (long-term) storage. Otherwise, they will only last until the module loses power.
CN	Exit command mode now. (If you don't send any commands to the module for a few seconds, command mode will timeout and exit even without a CN command.)

API mode

As an alternative to Transparent Operation, API (Application Programming Interface) Operations are available. API operation requires that communication with the module be done through a structured interface (data is communicated in frames in a defined order). The API specifies how commands, command responses and module status messages are sent and received from the module using a UART Data Frame.

Read the manual if you are going to use the API mode.

Jumper setting

The XBee shield has two jumpers (the small removable plastic sleeves that each fit onto two of the three pins labelled Xbee/USB). These determine how the XBee's serial communication connects to the serial communication between the microcontroller (ATmega8 or ATmega168) and FTDI USB-to-serial chip on the SainSmart board.

With the jumpers in the XBee position (i.e. on the two pins towards the interior of the board), the DOUT pin of the XBee module is connected to the RX pin of the microcontroller; and DIN is connected to TX. Note that the RX and TX pins of the microcontroller are still connected to the TX and RX pins (respectively) of the FTDI chip - data sent from the microcontroller will be transmitted to the computer via USB as well as being sent wirelessly by the XBee module. The microcontroller, however, will only be able to receive data from the XBee module, not over USB from the computer.

With the jumpers in the USB position (i.e. on the two pins nearest the edge of the board), the DOUT pin the XBee module is connected to the RX pin of the FTDI chip, and DIN on the XBee module is connected to the TX pin of the FTDI chip. This means that the XBee module can communicate directly with the computer - however, this only works if the microcontroller has been removed from the SainSmart board. If the microcontroller is left in the SainSmart board, it will be able to talk to the computer normally via USB, but neither the computer nor the microcontroller will be able to talk to the XBee module.

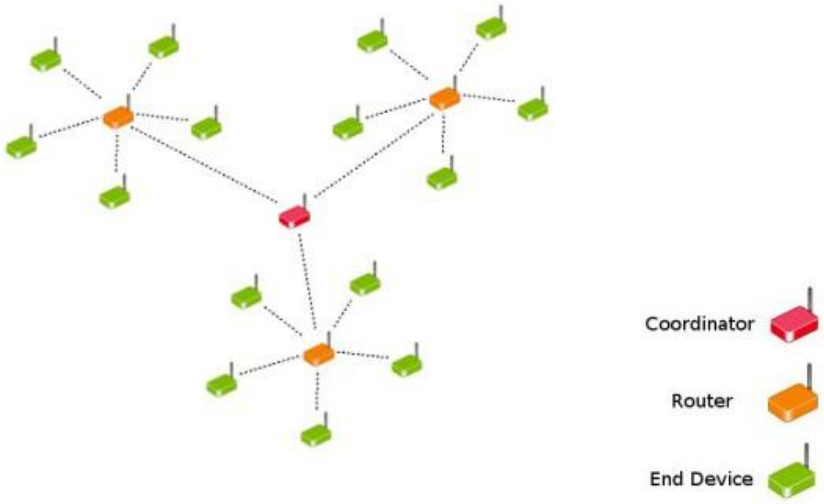
Using Series 2 ZB XBee's

Series 2 XBee's (ZigBee protocol) are quite different to 802.15.4 ones.

ZigBee networks are called personal area networks or PANs. Each network is defined with a unique PAN identifier (PAN ID). XBee ZB supports both a 64-bit (extended) PAN ID and a 16-bit PAN ID.

The 16-bit PAN ID is used in all data transmissions. The 64-bit PAN ID is used during joining, and to resolve 16-bit PAN ID conflicts that may occur.

ZigBee defines three different device types: coordinator, router, and end devices.



Coordinator

Selects a channel and PAN ID (both 64-bit and 16-bit) to start the network
Can allow routers and end devices to join the network
Can assist in routing data
Cannot sleep--should be mains powered.

Router

Must join a ZigBee PAN before it can transmit, receive, or route data
After joining, can allow routers and end devices to join the network
After joining, can assist in routing data
Cannot sleep--should be mains powered.

End device

Must join a ZigBee PAN before it can transmit or receive data
Cannot allow devices to join the network
Must always transmit and receive RF data through its parent. Cannot route data.
Can enter low power modes to conserve power and can be battery-powered.

In ZigBee networks, the coordinator must select a PAN ID (64-bit and 16-bit) and channel to start a network. After that, it behaves essentially like a router. The coordinator and routers can allow other devices to join the network and can route data.

After an end device joins a router or coordinator, it must be able to transmit or receive RF data through that router or coordinator. The router or coordinator that allowed an end device to join becomes the "parent" of the end device. Since the end device can sleep, the parent must be able to buffer or retain incoming data packets destined for the end device until the end device is able to wake and receive the data.

Chapter 23: MPU-6050 Sensor

What is the MPU-6050 Sensor?

The MPU-6050 sensor has both a 3-axis gyroscope and a 3-axis accelerometer combined together on the same silicon die, as well as an onboard Digital Motion Processor™ (DMP™). The MPU-6050 IC, along with a 3~5v input power regulator, supporting discrete, interface parts (e.g., pull-up resistors for I2C bus, etc), are co-located on a breadboard friendly breakout PCB, as shown in Figure 23.1. From this figure you can see the markings for the X-axis (pointing right) and the Y-axis (pointing up). The Z-axis is orthogonal (at a right angle) to both of these axes, so it is pointing straight out from the surface of the PCB.

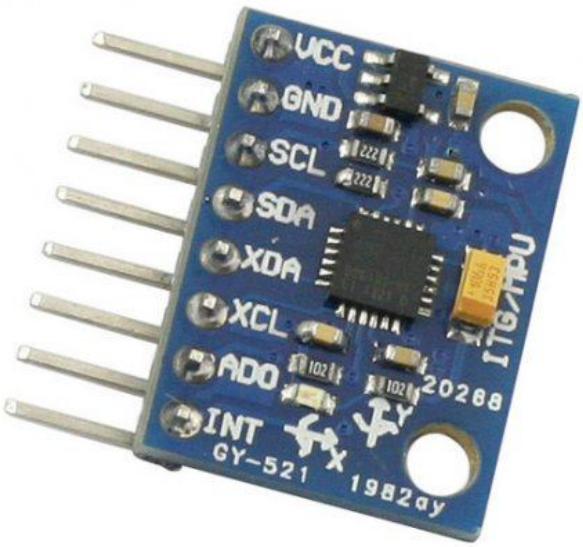


Figure 23.1 Blown up view of the MPU-6050 sensor.

Each axis of the gyroscope and accelerometer has its own 16-bit analog-to-digital converter associated with it. Thus, the analog values produced by each axis are converted to digital, 16-bit words. This "raw" data can be used by the onboard DMP, or it can be sent over the I2C bus to an external processor, such as the Uno.

The I2C (inter-integrated circuit) uses bidirectional communications via two open-drain lines: the Serial Data Line (SDA), and the Serial Clock Line (SCL). In

using the Uno with the MPU-6050 sensor, the Uno acts as the Master of the I2C bus and the MPU-6050 acts as a slave. Thus, the MPU-6050 only sends data when requested by the Uno. The Uno uses analog pin 5 for SCL and analog pin 4 for SDL.

The MPU-6050 also has a secondary I2C interface, where it is the Master and an external magnetometer or similar would be the slave. The use of a magnetometer would add directional information, which could be used by the MPU-6050's DMP.

The sensitivity of the gyroscope and the accelerometer is done through setting the appropriate registers and bits in the MPU-6050. The possible sensitivity settings are as follows:

Description
Gyroscopes range: +/- 250, 500, 1000, 2000 degree/sec
Acceleration range: +/- 2g, +/- 4g, +/- 8g, +/- 16g

If you need more information about the MPU6050, visit:
http://www.sainsmart.com/sainsmart-mpu-6050-3-axis-gyroscope-module.html

Raw acceleration and gyro data display sketch

Sketch components
- 1 x MPU-6050 sensor module
- Uno, breadboard, & jumper wires

Connections
Only 4 wires are required to operate the MPU-6050 sensor module with the Uno: two wires for the I2C bus, and two wires for power/ground. Table 23.1 lists these connections.

MPU-6050 Module Pin	Uno Pin
Vcc	5v or 3.3v
GND	GND
SCL	Analog pin 5
SDL	Analog pin 4

Table 23.1 Wire connections for sketch.

Required files for the sketch
The code for this sketch can be uploaded to your Uno from the SainSmart library. The sketch: *mpu6050_raw.ino* needs an additional, special library named *MPU6050* to compile and work properly. The files in this library have two types of functions: for I2C communication, and for MPU_6050 initialization and access. The original files needed for the library *MPU6050* are located at the following github links:

https://github.com/jrowberg/i2cdevlib/tree/master/Arduino/MPU6050

https://github.com/jrowberg/i2cdevlib/tree/master/Arduino/I2Cdev

They are also located under the SainSmart Website at:

http://www.sainsmart.com/zen/documents/20-011-926/MPU-6050/GY521mpu-6050/code/Arduino/MPU6050/

The *MPU6050* library is also included under the SainSmart code for Chapter 23.

Copy the *MPU6050* directory to the arduino IDE's *libraries* folder, located under the folder your arduino program is installed (e.g., Linux: ~/arduino-1.0.5/libraries/ or Windows: > Computer > Local Disk(C:) > Program Files (x86) > Arduino > libraries). If you have the Arduino IDE already up before you install this library, you will need to close it and re-launch it.

Sketch description

Once you have upload the *mpu6050_raw* sketch to the Uno, open the Serial Monitor window by clicking on the button in the upper right hand corner of the Arduino IDE using your mouse. Make sure the baud rate in the lower right hand corner of this window is set to 38400.

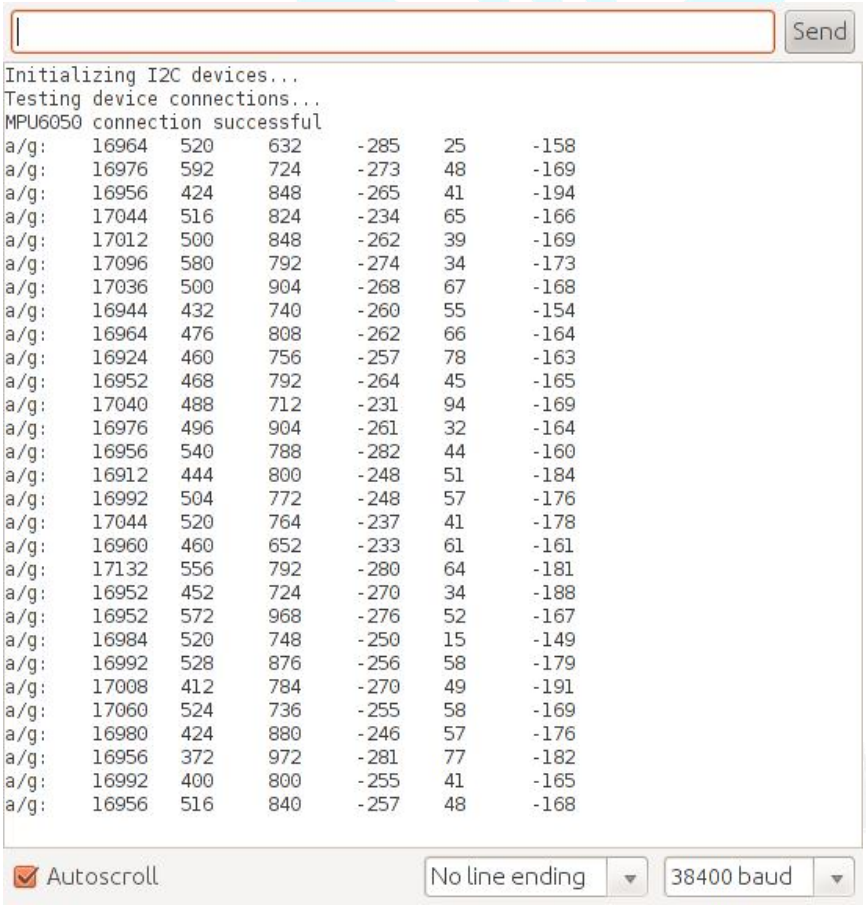


Figure 23.2 Example raw data from the *mpu6050_raw* sketch.

As can be seen from Figure 23.2, the data is a bit hard to interpret. In looking at the sections of code that acquires and prints out the acceleration and gyro values obtained from the MPU-6050:

// read raw accel/gyro measurements from device

```
    accelgyro.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);

// display tab-separated accel/gyro x/y/z values
    Serial.print("a/g:\t");
    Serial.print(ax); Serial.print("\t");
    Serial.print(ay); Serial.print("\t");
    Serial.print(az); Serial.print("\t");
    Serial.print(gx); Serial.print("\t");
    Serial.print(gy); Serial.print("\t");
    Serial.println(gz);
```

We can see that the high level for acceleration in the X-axis is much higher than the values in the Y-axis & Z-axis. That is because the X-axis is orthogonal to the surface of the earth. Thus, gravity is causing an acceleration along the X-axis. If you move the sensor/breadboard so that the Y-axis is orthogonal to the earth's surface, then that value will be much higher. Same with the Z-axis. Additionally, while moving the sensor around, you will also see the gyro's axes values changing, indicating change in movement.

Unfortunately, the makers of this chip are very guarded with information on how to use the DMP. Using this would make it much easier to get more definitive data. However, there are people out there that have developed arduino code that implements a Kalman filter, which does produce great results.

Chapter 24: Keypad

The keypad that comes with one of the SainSmart Uno Starter Kits (pictured in Figure 24.1) is of a matrix, pushbutton style. Thus, the keypad can be viewed as having four rows and four columns, see Figure 24.2. A connection is made between the corresponding row line and column line when a button is pressed, which is typically used to complete a circuit. So, when none of the buttons are pressed, there are no connections between the row and column lines (all open circuits).

Figure 24.1 Matrix style keypad.

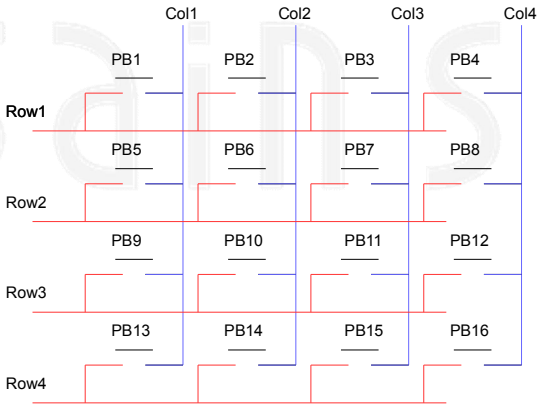


Figure 24.2 Circuit representation of matrix style keypad.

Keypad sketch

Sketch components

- 1 x Keypad
- Uno, breadboard, & jumper wires

Connections

The connections for this sketch are very straight forward. Looking at Figure 24.1, the first 4 pins from the left side represent rows 1 thru 4, and the remaining 4 pins represent columns 1 thru 4. So the sketch has been written so that pin 1 (left most pin) of the key pad connect to the Uno's digital I/O pin 9, and the remaining pins connect in succession, so that pin 8 of the keypad (right most pin) connects to the Uno's digital I/O pin 2. This is also defined in the top part of the sketch, as shown in Figure 24.3, under the array of bytes: rowPins[], and colPins[]. If your Uno has the extra three row headers (S V G), then you can simply plug the keypad's connecter into the S row, with the pins lining up as just mentioned.

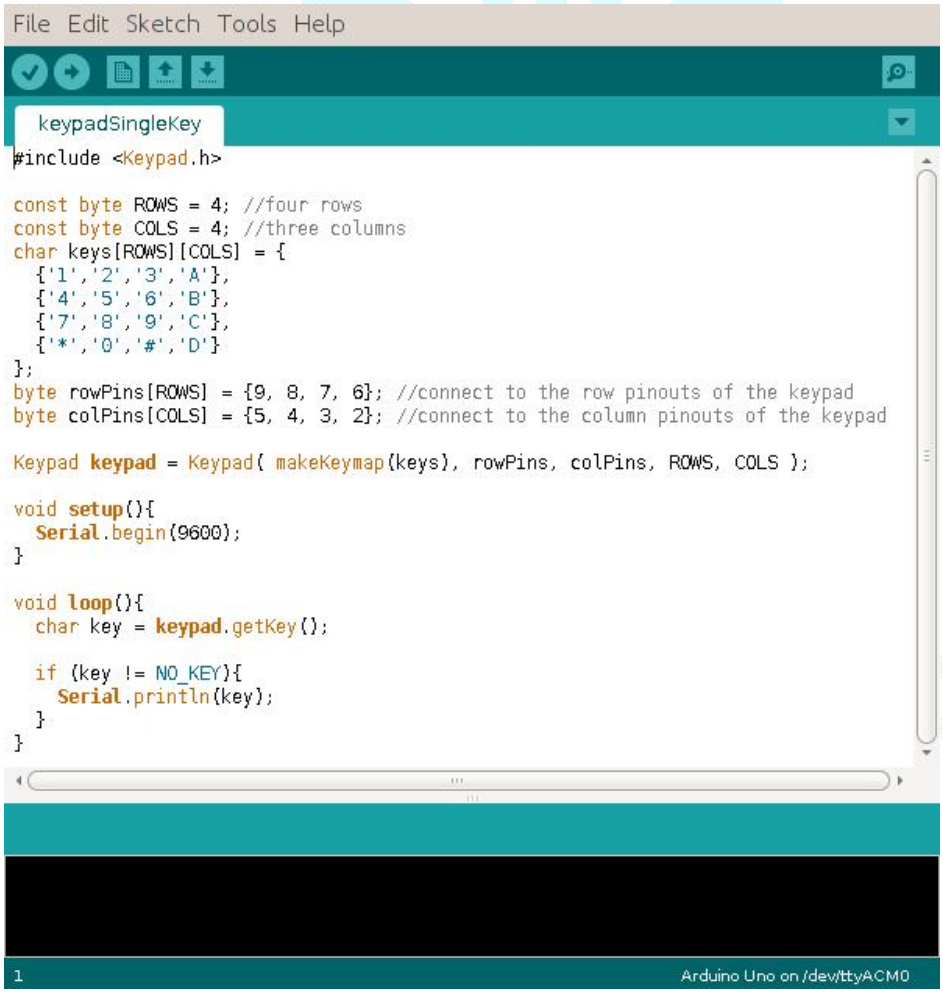


Figure 24.3 Keypad sketch.

Required files for the sketch

The code for this sketch can be uploaded to your Uno from the SainSmart library

under Chapter 24 Keypad. The sketch: *keypadSingleKey.ino* needs an additional, special library named *Keypad* to compile and work properly (see the #include <Keypad.h> line of code at the very top of Figure 24.3). The files in this library have functions that simplify the checking and retrieving keypad presses. The original files needed for the library *Keypad* are located under the Arduino Website at:

http://playground.arduino.cc/Code/Keypad

This library is also included under the Chapter 24 code from SainSmart.

Copy the *Keypad* directory to the arduino IDE's *libraries* folder, located under the folder your arduino program is installed (e.g., Linux: ~/arduino-1.0.5/libraries/ or Windows: > Computer > Local Disk(C:) > Program Files (x86) > Arduino > libraries). If you have the Arduino IDE already up before you install this library, you will need to close it and re-launch it.

Sketch description

Once you have upload the *keypadSingleKey* sketch to the Uno, open the Serial Monitor window by clicking on the button in the upper right hand corner of the Arduino IDE using your mouse. Make sure the baud rate in the lower right hand corner of this window is set to 9600, as shown in Figure 24.4.

As you press the different keys on the keypad, they will be displayed on their own line, as show in Figure 24.4.

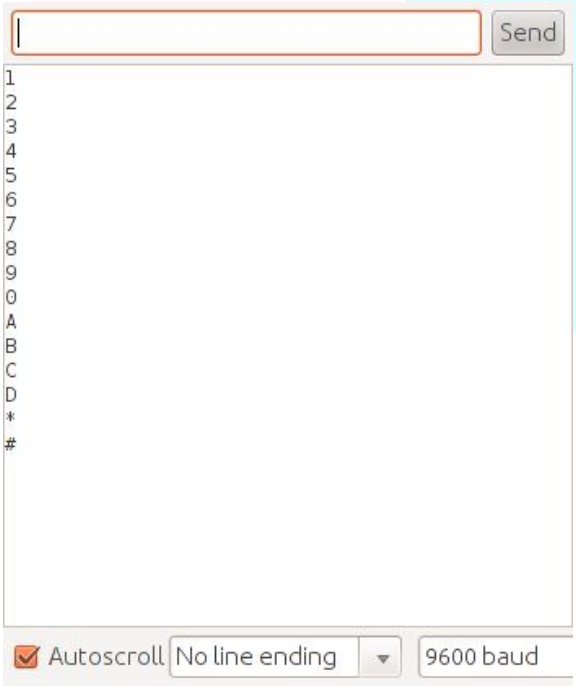


Figure 24.4 Sketch output in the Serial Monitor window.